Database Design and Implementation
Chapter 5 Conceptual Exercises

Questions	Answers
5.1.	1. **User A** initiates the seat reservation process. They are at the point right after **Step 1**, where the number of available seats has been retrieved and is stored in **numAvailable**. 2. Before **User A** proceeds to **Step 2**, **User B** initiates the seat reservation process. **User B** also completes **Step 1**, retrieving the same **numAvailable** as **User A** since **User A** has not yet updated the seat count. 3. Now, both **User A** and **User B** have the same **numAvailable** value, and both see at least one seat available. 4. **User A** proceeds to **Step 2** and updates the seat availability, decrementing the number by 1. 5. Before **User A** moves to **Step 3**, **User B** reaches **Step 2** with the stale **numAvailable** value and also decrements the seat count by 1. 6. Both **User A** and **User B** have now successfully reserved a seat (assuming there was at least two to begin with), and the **SEATS** table now reflects two fewer available seats. 7. **User A** reaches **Step 3** and updates their balance. However, if the result set for **User B** is retrieved before **User A**'s balance update and cached or held in the local memory, **User B** might retrieve the old balance amount. 8. **User B** proceeds to update the customer balance but with the stale balance information, therefore, only incrementing the balance by one seat price, not accounting for **User A**'s transaction. The final result is that the **SEATS** table shows two fewer seats, but the **CUST** table only reflects the price of one seat being added to the balance due because of the stale data read by **User B**. To prevent this, the operation should be enclosed in a transaction with proper isolation level to ensure atomic updates. Additionally, optimistic or pessimistic locking mechanisms could be used to prevent other operations from intervening between the check and the update.
5.2.a.	In configuration management systems like Git or Subversion, a transaction can be thought of as a set of changes or updates made to the files in a repository. It involves recording the before-and-after states of the repository so that these changes can be rolled back if necessary. It is similar to a transaction in a database system but tailored to version control, where the currency is not data records but files and their changes over time. A transaction's purpose in version control is to ensure atomicity and consistency. Changes are either applied as a whole (atomicity) or not at all, and they do not disrupt the functioning of the system (consistency). Since transactions in version control systems are about managing file states over time, they must provide a mechanism to track changes in a way that allows

	reverting to previous states. This involves recording not just the changes but also metadata like who made the change and when.
5.2.b.	In the context of version control, serializability refers to the property that ensures that concurrently executed transactions result in a system state that would be obtained if the transactions were executed serially. These systems use mechanisms such as locking, branching, and merging. Locking prevents concurrent access to the same resource. Branching allows multiple users to work independently on copies of the files. Merging incorporates changes from different branches back into a single version. Branching and merging, particularly in Git, enable a form of serializability by allowing changes to be integrated in a controlled manner. Merge conflicts may arise, which requires manual resolution to ensure that the system's state is consistent.
5.2.c.	Database systems already use transactions to manage concurrency and ensure atomicity, consistency, isolation, and durability (ACID properties). Database transactions are designed to handle complex data relationships and ensure that all transactions are ACID-compliant. While some principles are shared, the mechanisms are different due to the different types of data and operations involved. Locking mechanisms in databases are often more granular and sophisticated due to the complexity of data relationships.
5.3.a.	Even if a program is read-only, transactions can still be important to ensure that the read operations see a consistent state of the database. Transactions provide a 'snapshot' of the database, ensuring that all reads within the transaction see the same state. Without transactions, a read-only program might see intermediate states of the database if other transactions are writing data, leading to inconsistent or unexpected results.
5.3.b.	Running the entire read-only program as a single transaction could lock the database for an extended period, preventing other transactions from executing, leading to performance issues and decreased concurrency. Resource locking for a long duration can lead to database contention. Long transactions hold resources for more time than necessary, potentially delaying write transactions from other users.

Database Design and Implementation
Chapter 5 Conceptual Exercises

5.3.c.	The overhead of committing a read-only transaction is typically low, as there are no changes to be written back to the database. The overhead consists mainly of releasing any locks held and logging the commit. It may not be necessary to commit after every SQL query unless each query needs a different consistent view of the database.
5.4.a.	Start records in the log are crucial for identifying when a transaction begins, aiding in the recovery process. They provide a clear demarcation of the beginning of a transaction, which is essential during the recovery process to know which transactions were active at the time of a crash. With start records, the recovery manager can easily determine which transactions need to be rolled back or redone to restore the database to a consistent state.
5.4.b.	If a database system does not write start records, the recovery manager can still function but with limited capability. Without start records, it becomes difficult to ascertain the start of a transaction, which complicates the recovery process. The system might rely more on commit and abort records to infer transaction boundaries, which may not always be accurate or efficient, potentially leading to improper recovery operations.
5.5.	Writing the rollback log record to disk before the rollback method returns is a protocol adherent to the write-ahead logging (WAL) principle. The WAL principle dictates that no data modifications should be written to the database until all associated log records are safely stored. This ensures that the database can be recovered to a consistent state in the event of a system crash. If the rollback log is not written to disk and the system crashes, the database might not be able to undo the transactions correctly during recovery, potentially leaving the database in an inconsistent state. Ensuring that the rollback log record is written to disk enhances the robustness of the recovery process, making it a good practice. This guarantees that even if a crash occurs immediately after the rollback, the actions can be reversed to maintain database integrity.
5.6.	Not writing rollback log records when a transaction is rolled back deviates from the standard approach of logging all changes to the database, including rollbacks.

	Without rollback log records, there would be no direct traceability in the log for rollback actions taken, making it challenging to verify the actions performed during the recovery process. During recovery, the absence of rollback records could complicate the identification of the transactions that were rolled back, potentially leading to an incorrect recovery state. Choosing not to write rollback log records might reduce the immediate I/O overhead during rollback operations. However, it compromises the ability to audit the database and maintain accurate recovery mechanisms. The trade-off could lead to a less reliable system, and thus it is typically not a recommended practice. In systems where durability and reliability are paramount, it is essential to log all changes, including rollbacks.
5.7.	1. Flush the transaction's modified buffers to disk. 2. Write a commit record to the log. 3. Flush the log page containing the commit record. The algorithm for committing a transaction, as shown, is carefully ordered to maintain the ACID properties of database transactions, specifically durability and atomicity. Flushing modified buffers to disk ensures that all changes made by the transaction are durably stored before the commit is recorded. If we write the commit record before flushing buffers, there is a window where the commit record could be on disk, but the actual data changes are not. In case of a crash after the commit record is written but before the data is flushed, the system would consider the transaction committed despite its changes not being fully persisted. Swapping the steps would violate the durability property, as there would be no guarantee that all transaction changes survive a crash once a commit is logged, which is incorrect.
5.8.	Redoing a rollback or a recovery is based on the idempotence property of these operations. Idempotence means that applying the rollback or recovery operation multiple times has the same effect as applying it once. If the system crashes during rollback or recovery, redoing the same operations will not further change the state beyond what was intended by the original

	rollback or recovery action. This ensures the consistency and integrity of the database.
5.9.	Logging changes during rollback or recovery is generally necessary for ensuring recoverability. During normal operations, logging ensures that any changes can be undone or redone for recovery purposes. Logging during rollback and recovery operations ensures that if a failure occurs during these processes, the system can continue the rollback or recovery from the last logged operation. It contributes to the robustness of the system.
5.10.a.	The recovery algorithm would use the checkpoint of the oldest active transaction to minimize the recovery time by ignoring transactions that started after the checkpoint. When a checkpoint is initiated, the system logs the oldest active transaction and its state. During recovery, the system would need to consider only the transactions that were active at the time of the checkpoint or started afterward, which simplifies the rollback procedures.
5.10.b.	The mentioned strategy could be simpler to implement as it involves tracking only one transaction. However, it may not be as efficient since it could lead to longer recovery times if the oldest transaction is much older than others. Tracking a single transaction is straightforward, reducing complexity. Efficiency could suffer as more transactions might need to be considered in recovery, increasing the time taken.
5.11.	If the rollback method encounters a quiescent checkpoint log record, it should not affect the rollback process since it indicates a consistent state. This record implies that at the time of the checkpoint, there were no active transactions, and the system was in a consistent state. For a nonquiescent checkpoint, rollback must consider the state of the transactions as of the last checkpoint and undo changes of transactions that were active at the checkpoint but not completed. Upon encountering either type of checkpoint, rollback should continue to undo changes until it reaches the beginning of the transaction it's rolling back, using the checkpoint as a reference for the system's state at a known good point.

Database Design and Implementation
Chapter 5 Conceptual Exercises

5.12.	The restriction that new transactions cannot start while the checkpoint record is being written is important for ensuring a consistent view of the system at the time of the checkpoint. A checkpoint aims to provide a snapshot of all ongoing transactions at a certain point in time so that the system knows which transactions may need to be rolled back or redone during recovery. If new transactions were allowed to start during the checkpointing process, they might not be included in the checkpoint record. This could lead to a situation where the checkpoint record does not accurately reflect the system's state, potentially compromising the correctness of recovery operations. To ensure that recovery processes have a complete and accurate list of transactions to consider, it is necessary to prevent the start of new transactions while the checkpoint is being written.
5.13.a.	The two-record strategy allows checkpointing to start without immediately needing the complete list of active transactions, thereby avoiding a delay in the initiation of new transactions. The <BEGIN_NQCKPT> record marks the initiation of a checkpoint. It establishes a cut-off point for including transactions in the checkpoint without delaying the system. The second record, which contains the list of active transactions, is created after the list is fully determined. This allows for the accurate capture of all active transactions without blocking new transactions from starting after the <BEGIN_NQCKPT> record is written.
5.13.b.	The recovery algorithm must now account for the two types of checkpoint records: When the recovery process starts, it should look for the most recent <BEGIN_NQCKPT> record to establish the beginning of the checkpoint interval. The recovery process should then use the subsequent <NQCKPT ...> record to determine which transactions were active and need to be considered for rollback or redo during recovery.
5.14.	Quiescent checkpoints imply a stable state where no transactions were active. The recovery manager is designed to look backward from the point of failure to find the most recent checkpoint. Since a quiescent checkpoint indicates a stable system state, subsequent checkpoints do not overwrite but rather append to the log.

	The recovery process only needs to go back as far as the most recent consistent state, which is provided by the latest quiescent checkpoint. Hence, it disregards any checkpoints that occurred before it.
5.15.	It is possible to encounter several nonquiescent checkpoint records because they can occur more frequently and do not require the system to be quiescent. Example: If a system takes nonquiescent checkpoints at regular intervals, and transactions are long-running, several such checkpoints could occur during the lifetime of a single transaction. Handling Subsequent Nonquiescent Checkpoints: The best way to handle multiple nonquiescent checkpoints is to always consider the most recent one since it includes all the transactions of previous nonquiescent checkpoints and possibly more.
5.16.	The system is designed such that a nonquiescent checkpoint turns into a quiescent checkpoint only when all transactions have completed. Thus, once a quiescent checkpoint is established, there can be no nonquiescent checkpoint after it until a new transaction starts. A nonquiescent checkpoint becomes quiescent only when all transactions at the time of the checkpoint have finished. Since a quiescent checkpoint represents a point where the system had no active transactions, any checkpoint found after this point in the log must be of a new series of transactions, thus making it impossible to have both a nonquiescent and a quiescent checkpoint for the same set of transactions.
5.17.a.	A transaction that has been rolled back has already undone its changes to the database, so its effects are not present in the current state of the database. The log reflects all changes, including the undo actions taken during the rollback. Therefore, any changes that a rolled-back transaction made have been counteracted by this point. Since the rolled-back transactions' effects have been nullified, the undo stage of recovery does not need to address them again, making it correct to bypass them in step 1c.
5.17.b	If the algorithm were to undo the values for transactions that have already been rolled back, it would introduce inconsistency.

	Reversing the actions of an already rolled-back transaction could potentially revert the corrective actions taken during the rollback, leading to an incorrect database state. Such an action would undo the rollback, thereby applying the effects of a transaction that was deemed invalid or needed to be reversed, which is logically incorrect.
5.18.	Rollback actions typically have priority and are executed in a controlled environment, where the transactional integrity must be preserved above concurrent operation considerations. When a rollback occurs, it is often because of an aborted transaction or a recovery process after a failure. In such cases, maintaining database integrity is paramount. The rollback is correcting errors or undoing incomplete work. Locking mechanisms are designed to manage concurrency for operational transactions, not for transactions that are being terminated or reversed. Since the rollback operation is supposed to restore a previous state, it would not lead to a non-serializable conflict because any active transactions should not be accessing the data being rolled back, as it's not in a committed state.
5.19.	Undo-only and redo-only recovery are designed for different scenarios, and their information requirements do not overlap in a way that would allow a combination of the two approaches. Undo recovery is necessary for transactions that have not committed and need to be reversed. Redo recovery is for transactions that have committed but their effects have not been fully reflected in the database due to system failure. Undo information includes the original state before transaction operations, necessary to roll back changes. Redo information contains the intended state after transaction operations, necessary to complete the committed transactions. Each type of recovery requires the presence of specific information that the other does not provide. Without undo information, it’s not possible to reverse a transaction, and without redo information, it’s not possible to reapply committed changes. Thus, a recovery algorithm must maintain both undo and redo information to handle the range of possible recovery scenarios.
5.20.a.	• This approach involves two phases: Undo and Redo.

	• During the undo phase, all transactions that had started but not committed at the time of the crash are rolled back. • During the redo phase, all transactions that had committed are replayed to ensure that their changes are reflected in the database. **Steps**: 1. **Redo Phase**: • Transaction T2 has a commit record; thus, its changes will be redone. • Transaction T3 does not have a commit record; it will be ignored in this phase. • Transaction T4 has a commit record; thus, its changes will be redone. 2. **Undo Phase**: • Transaction T1 does not have a commit or abort record, implying it was active at the time of crash; it will be undone. • Transaction T3 was active and did not commit; it will be undone. **Redo actions**: • Apply **<SETSTRING, 2, junk, 33, 0, abc, def>** (Redo T2) • Apply **<SETSTRING, 4, junk, 55, 0, abc, sue>** (Redo T4) • Apply **<SETSTRING, 4, junk, 55, 0, sue, max>** (Redo T4) **Undo actions**: • Reverse **<SETSTRING, 1, junk, 44, 0, abc, xyz>** (Undo T1) • Reverse **<SETSTRING, 3, junk, 33, 0, def, joe>** (Undo T3) **Changes to the database**: • Since T2 and T4 committed, their changes will persist. • Since T1 and T3 did not commit, their changes will not persist.
5.20.b.	Undo-only recovery: **Analysis**: • This approach involves only rolling back the transactions that had started but not committed at the time of the crash. **Steps**: 1. **Undo Phase**: • Only consider transactions without commit records. **Undo actions**: • Reverse **<SETSTRING, 1, junk, 44, 0, abc, xyz>** (Undo T1) • Reverse **<SETSTRING, 3, junk, 33, 0, def, joe>** (Undo T3) **Changes to the database**: • Since T1 and T3 did not commit, their changes will not persist. • Changes made by T2 and T4 remain because they committed, even though the system does not explicitly redo them after the crash.
5.20.c.	It is not possible for T1 to have committed based on the given log because there is no **<COMMIT, 1>** record present.

Database Design and Implementation
Chapter 5 Conceptual Exercises

5.20.d.	It is possible for T1 to have modified a buffer containing block 23 in memory, but it is not recorded in the log provided. Since the log does not mention block 23, we cannot conclude from the given log whether it was modified or not
5.20.e.	It is not possible for T1 to have modified block 23 on disk because the log records only show modifications to block 44 for T1, and without a commit record, changes would not have been flushed to the disk.
5.20.f.	It is not possible for T1 to have not modified a buffer containing block 44 since the log explicitly shows a modification operation on block 44 by T1.
5.21.	A serial schedule is always serializable because it is a specific type of serializability where transactions are run completely one after another, without any interleaving of operations from different transactions. A serializable schedule is not always serial. Serializable means the effects of the schedule are the same as if the transactions had been executed in some serial order, but the transactions themselves may be interleaved as long as their interleaving does not violate the consistency of the database.
5.22.a.	If the database is much larger than the buffer pool: • Concurrency allows for the utilization of I/O and CPU concurrently, improving throughput. • While one transaction is waiting for I/O, another can use the CPU to proceed with its computation, leading to better resource utilization.
5.22.b.	If the database fits into the buffer pool: • Concurrency is less important because I/O wait times are significantly reduced or eliminated. • The system can quickly switch between transactions without the overhead of disk I/O, so the benefits of concurrency are diminished.
5.23.	The get/set methods in the SimpleDB class Transaction do not unlock the block when done because: • Prolonged control over the block is necessary to maintain isolation between transactions. • Locks are typically held until the transaction completes (commits or aborts) to prevent other transactions from making
5.24.	The history of the transactions in Figure 5.3 involves multiple steps where a file is accessed and modified by transactions. Here's a breakdown: • **tx1** pins the block, writes integers and strings without logging (assuming no other concurrent transactions interfere at this point). • **tx1** commits, making its changes to the block permanent. • **tx2** starts, pins the same block, reads values, modifies them, and commits. Since files are the concurrency element, tx2 would wait if tx1 had not committed yet. • **tx3** starts, repeats the reading, and modifies the data at location 80 but then rolls back, so the changes made by tx3 to location 80 are undone.

	• **tx4** reads the value at location 80 to confirm that the rollback was successful. If files are the element of concurrency, the system ensures that when one transaction is actively writing to a file, no other transaction can read or write to the same file until the first transaction commits or rolls back.
5.25.a.	A serializable schedule is one that can be transformed into a serial schedule (transactions executed sequentially without overlapping) through some series of swaps of non-conflicting operations. • First, let's lay out the operations of both transactions as they are provided: T1: W(b1) R(b2) W(b1) R(b3) W(b3) R(b4) W(b2) T2: R(b2) R(b3) R(b1) W(b3) R(b4) W(b4) • To create a non-serial, but serializable schedule, we can interleave the operations of T1 and T2, ensuring that we do not violate the serializability. A simple method to maintain serializability is to make sure we respect the read and write dependencies between the transactions. • Given this, one possible non-serial serializable schedule could be: T2: R(b2) T1: W(b1) T2: R(b3) T1: R(b2) T1: W(b1) T2: R(b1) T1: R(b3) T1: W(b3) T2: W(b3) T1: R(b4) T2: R(b4) T2: W(b4) T1: W(b2) This schedule is equivalent to T1 followed by T2, because even though operations are interleaved, the outcome is as if T1 completed all its operations before T2 began.
5.25.b.	For this part, we need to ensure that we follow the two-phase locking protocol, which means that a transaction must obtain all the locks it needs before it releases any lock. Locks are applied before a read (shared lock) or write (exclusive lock) operation and released after the transaction has finished its operations on the data item. Initial Reasoning for T1 and T2: • Lock (X or S) before Read/Write. • Unlock after all operations on that data item are done.

	Step 1: T1 1. X-lock b1, Write b1, X-lock b2, Read b2, Write b1, Unlock b1. 2. Read b3, X-lock b3, Write b3, Unlock b3. 3. Read b4, X-lock b2 (because b2 will be written), Write b2, Unlock b2. 4. Unlock b4 after Read since it's the last operation for T1. Step 2: T2 1. S-lock b2, Read b2, Unlock b2 after T1 is done with b2. 2. S-lock b3, Read b3, X-lock b3 (T1 done with b3), Write b3, Unlock b3. 3. S-lock b1, Read b1 (after T1 releases b1). 4. Read b4, X-lock b4, Write b4, Unlock b4. Adjusted Step 1: T1 with Locking Protocol • X-lock b1, Write b1. • X-lock b2, Read b2, Write b1, Unlock b1. • X-lock b3, Read b3, Write b3, Unlock b3. • Read b4, Write b2, Unlock b2. • Unlock b4. Adjusted Step 2: T2 with Locking Protocol • S-lock b2, Read b2. • S-lock b3, Read b3. • S-lock b1, Read b1, Unlock b1. • Upgrade S-lock b3 to X-lock, Write b3, Unlock b3. • Read b4, X-lock b4, Write b4, Unlock b4. • Unlock b2 after b2 is written in T1.
5.25.c	To create a non-serial schedule that leads to a deadlock, we can have the following interleaved schedule: 1. T1: W(b1) 2. T2: R(b2) 3. T1: R(b2) (T1 is blocked here waiting for T2 to release the lock on b2) 4. T2: R(b3) 5. T1: W(b1) 6. T2: R(b1) (T2 is blocked here waiting for T1 to release the lock on b1) At this point, T1 and T2 are both blocked and waiting for locks held by the other transaction, resulting in a deadlock.
5.25.d	To show that there is no non-deadlocked non-serial serializable schedule for these transactions that obeys the lock protocol, we can use the fact that T1 and T2 both need to read and write to b1 and b2, and there's a circular dependency between these locks. The lock protocol prevents a transaction from releasing a lock until it has completed its operations. Therefore, any interleaved schedule of

	T1 and T2 that obeys the lock protocol will eventually lead to a deadlock due to the circular dependency between locks b1 and b2.
5.26.	To create a serializable schedule with conflicting write-write operations that do not affect the order in which transactions commit, consider the following transactions: T1: W(x); R(y) T2: R(x); W(y) Here's a possible schedule: 1. T1: W(x) 2. T2: R(x) 3. T2: W(y) 4. T1: R(y) In this schedule, T1 and T2 have conflicting write-write operations on variables x and y, respectively. However, the final result is still serializable because the order in which transactions commit is preserved.
5.27.	If all transactions obey the two-phase locking protocol, then all schedules are serializable. This is because the two-phase locking protocol ensures that a transaction acquires and releases locks in two phases: 1. The growing phase: In this phase, a transaction can acquire locks but cannot release any locks. 2. The shrinking phase: In this phase, a transaction can release locks but cannot acquire any new locks. By following this protocol, transactions guarantee that they will not release any locks until they have acquired all the locks they need. This prevents any cycles in the precedence graph, ensuring serializability.
5.28.	The waits-for graph has a cycle if and only if there is a deadlock. In the waits-for graph, each node represents a transaction, and a directed edge from transaction T1 to transaction T2 indicates that T1 is waiting for T2 to release a lock. If there is a cycle in this graph, it means that there is a chain of transactions waiting for each other, creating a circular dependency, which is the definition of a deadlock. Conversely, if there is no cycle in the waits-for graph, it implies that there are no circular dependencies, and therefore, there is no deadlock in the system. Transactions are not waiting indefinitely for each other to release locks, ensuring that the system is free from deadlocks.

Database Design and Implementation
Chapter 5 Conceptual Exercises

5.29.	Among the possibilities mentioned for handling deadlocks in a waits-for graph, rolling back the transaction whose request caused the cycle in the graph makes the most sense. Here's why: Rolling back the transaction that caused the cycle directly addresses the root cause of the deadlock. By doing so, you eliminate the source of contention and have a higher chance of allowing the remaining transactions to proceed and complete successfully. The other options, such as rolling back the oldest or newest transaction in the cycle or based on the number of locks held, may not necessarily address the actual cause of the deadlock. Rolling back an older transaction might not release the resource that is causing the deadlock, and rolling back a newer transaction might penalize a transaction that is not responsible for the deadlock. Rolling back the transaction that holds the most locks can also be problematic because it may not be the transaction that is directly causing the deadlock. Similarly, rolling back the transaction with the fewest locks may not resolve the deadlock if it's not the one causing the issue. In summary, rolling back the transaction that initiated the cycle in the waits-for graph is the most effective and targeted approach to resolve deadlocks.
5.30.	In SimpleDB, if transaction T currently has a shared lock on a block and calls **setInt** on it, it may cause a deadlock in a scenario where another transaction is waiting to obtain an exclusive lock on the same block. Here's a scenario: 1. Transaction A acquires an exclusive lock on block X. 2. Transaction B requests a shared lock on block X and is granted it because exclusive locks are incompatible with shared locks. 3. Transaction T requests an exclusive lock on block X but is blocked because both Transaction A and Transaction B have locks on the same block. 4. Transaction B decides to update the block by calling **setInt**, which requires an exclusive lock, but it cannot proceed because Transaction T is waiting for an exclusive lock on the same block. 5. Now, we have a deadlock: Transaction B is waiting for Transaction T to release the lock, and Transaction T is waiting for Transaction B to release the lock. This scenario demonstrates how a shared lock can lead to a deadlock when a subsequent request for an exclusive lock is made on the same resource.
5.31.	To create a schedule that causes a deadlock in the **ConcurrencyTest** class, consider the following transactions:

	Transaction A (represented by class A): • Pins two blocks: blk1 and blk2. • Requests a shared lock on blk1 and receives it. • Sleeps for 1 second to simulate some work. • Requests a shared lock on blk2 and receives it. • Commits the transaction. Transaction B (represented by class B): • Starts concurrently with Transaction A. • Creates a new transaction, txB. • Requests a shared lock on blk2. • Sleeps for a short time. • Requests a shared lock on blk1. Transaction C (represented by class C): • Starts concurrently with Transactions A and B. • Creates a new transaction, txC. • Requests a shared lock on blk1. • Sleeps for a short time. • Requests a shared lock on blk2. Now, let's analyze this scenario: 1. Transaction A pins and receives locks on both blk1 and blk2. 2. Transaction B starts and requests a lock on blk2 but is blocked because Transaction A is holding a shared lock on blk2. 3. Transaction C starts and requests a lock on blk1 but is blocked because Transaction A is holding a shared lock on blk1. 4. Transaction A completes its work and commits, releasing both locks. 5. Transaction B and Transaction C both acquire the locks they were waiting for (blk2 and blk1, respectively) and continue. At this point, Transaction B and Transaction C were initially blocked but were able to proceed once Transaction A released the locks. This leads to a situation where all transactions make progress without a deadlock.
5.32.	In this locking scenario described for the **ConcurrencyTest** class: • Initially, Transaction A (represented by class A) pins and receives locks on blk1 and blk2. • Transaction B (represented by class B) starts and requests a shared lock on blk2 but is blocked because Transaction A is holding a shared lock on blk2. • Transaction C (represented by class C) starts and requests a shared lock on blk1 but is blocked because Transaction A is holding a shared lock on blk1.

	At this point, a cycle has formed in the waits-for graph: 1. Transaction B is waiting for blk2 held by Transaction A. 2. Transaction C is waiting for blk1 held by Transaction A. 3. Transaction A is not waiting for any resource. This cycle in the waits-for graph indicates a potential deadlock. However, as described in the previous answer, the deadlock is resolved when Transaction A releases the locks, allowing Transactions B and C to proceed without any deadlock occurring.
5.33.a.	The wound-wait protocol prevents deadlock by ensuring that if an older transaction needs a lock held by a younger one, the older transaction "wounds" or aborts the younger one and takes the lock. This approach breaks any potential cycles in the waits-for graph, as older transactions always get priority. Since cycles in the waits-for graph are a sign of potential deadlock, the wound-wait protocol effectively prevents deadlocks.
5.33.b.	Wait-Die Protocol: In this protocol, older transactions are allowed to wait for locks held by younger transactions. Younger transactions requesting locks held by older ones are aborted. This minimizes the number of transaction restarts but can lead to starvation of younger transactions. Wound-Wait Protocol: In this protocol, older transactions are prioritized and can abort younger transactions if they request locks held by older ones. This ensures that older transactions make progress but can result in more frequent transaction restarts for younger transactions. The choice between the two protocols depends on the specific requirements of the system. Wait-die is suitable when it's crucial to avoid the frequent restart of younger transactions, while wound-wait is suitable when ensuring progress for older transactions is a higher priority, even if it means more restarts for younger transactions.
5.34.	Modifying the wait-die deadlock detection protocol to abort transactions if they request a lock held by a younger transaction would also detect deadlocks. However, this revised protocol has some important differences compared to the original one: Original Wait-Die Protocol: Aborts younger transactions to allow older transactions to proceed. It prioritizes the completion of older transactions and may lead to more frequent restarts of younger ones. Revised Protocol: Aborts transactions that request locks held by younger transactions. It prioritizes the completion of younger transactions over older ones and may lead to older transactions getting delayed or blocked.

	The choice between the two protocols depends on the system's priorities. The original wait-die protocol prioritizes the completion of older transactions, while the revised protocol prioritizes the completion of younger transactions. The preference for one protocol over the other depends on the specific requirements and performance goals of the system.
5.35.	The **lock** and **unlock** methods in class LockTable are synchronized to ensure thread safety when multiple threads concurrently attempt to acquire or release locks. Without synchronization, the following issues could occur: • Race Conditions: Multiple threads might simultaneously check the lock state and attempt to modify it. This can lead to inconsistent or incorrect behavior, such as multiple threads mistakenly thinking they have successfully acquired a lock when only one should have. • Incomplete Lock Acquisition: Without synchronization, a thread may think it has acquired a lock but is preempted by another thread before it can actually acquire it. This can result in a situation where a resource is not properly locked but is treated as if it is. • Inconsistent State: LockTable may become corrupted if multiple threads concurrently update its internal data structures, leading to undefined behavior and potential crashes. By synchronizing the **lock** and **unlock** methods, you ensure that only one thread can access and modify the lock state at a time, preventing these issues and ensuring correct lock management.
5.36.	Phantoms are not possible in a database system that uses files as concurrency elements because phantoms typically refer to a phenomenon that occurs with row-level or tuple-level concurrency control, not at the level of files or entire tables. Phantoms occur when a transaction sees a set of rows in one query and, before it commits, another transaction inserts, updates, or deletes rows that would affect the first transaction's query result. In a file-based concurrency control system, transactions typically acquire locks on entire files or sections of files, not individual rows or tuples. Therefore, there is no concept of phantoms because transactions deal with files as atomic units. Any updates to a file by one transaction are typically locked until that transaction completes, preventing other transactions from modifying the same file concurrently. Phantoms are more relevant in a context where fine-grained concurrency control is applied, such as row-level locking or snapshot isolation at the level of individual database rows.

Database Design and Implementation
Chapter 5 Conceptual Exercises

5.37.	Here's a high-level algorithm for deadlock detection that also handles transactions waiting for buffers: 1. Maintain a waits-for graph that represents the relationships between transactions and the resources (locks or buffers) they are waiting for. 2. Whenever a transaction requests a lock or a buffer and cannot acquire it immediately, add an edge to the waits-for graph indicating that the transaction is waiting for that resource. 3. Periodically, or when a new edge is added to the graph, check for cycles in the waits-for graph. Cycles represent potential deadlocks. 4. If a cycle is detected, identify the transactions involved in the cycle. These transactions are in a deadlock situation. 5. Choose a deadlock resolution policy, such as aborting one or more transactions involved in the cycle, to break the deadlock. 6. Rollback the selected transactions to release the resources they hold. 7. Retry the affected transactions or take appropriate actions based on the application's logic. This algorithm combines deadlock detection with handling transactions waiting for buffers, ensuring that both lock-related and buffer-related deadlocks can be identified and resolved.
5.38.	To rewrite the algorithm for multiversion locking so that the concurrency manager only makes one pass through the log file, you can implement a single-pass approach by considering the following steps: 1. Initialize a data structure to keep track of the versioned states of data items. This data structure can be a multiversion buffer or a similar construct. 2. Start reading the log file from the beginning. 3. For each log record encountered, update the versioned state of the corresponding data item based on the log record's information. 4. Continue processing log records sequentially, updating the versioned state of data items as necessary. 5. During this process, you can also maintain a waits-for graph and perform deadlock detection and resolution as needed. 6. Once you have processed all log records, the versioned states of data items will be up-to-date, and you can continue with normal transaction processing, including handling read and write requests from transactions. By processing the log file in a single pass and updating the versioned states of data items along the way, you can reduce the need for multiple iterations through the log file and improve efficiency in a multiversion locking system.

Database Design and Implementation
Chapter 5 Conceptual Exercises

5.39.	The read-committed transaction isolation level aims to reduce a transaction's waiting time by releasing shared locks (slocks) early. This approach benefits concurrency and responsiveness in scenarios where multiple transactions are accessing the same data concurrently. Here are the advantages of early lock release and illustrative scenarios:
	
	1. Improved Concurrency: By releasing shared locks immediately after a read operation, read-committed transactions allow other transactions to access the same data without waiting for the first transaction to complete. This improves concurrency and reduces contention.
	2. Reduced Blocking: Transactions holding shared locks do not block other transactions from reading the same data. This reduces the likelihood of transactions waiting for resources and potentially getting stuck in a deadlock situation.
	3. Shorter Transaction Duration: With early lock release, transactions can complete their read operations quickly and release locks promptly. This leads to shorter transaction durations, allowing more transactions to be processed in a given time frame.
	
	Illustrative Scenarios:
	• Scenario 1: Multiple Read-Only Transactions
	  • Suppose there are several read-only transactions concurrently accessing a database. In a read-committed isolation level, they can all read the same data simultaneously without waiting for each other, leading to improved throughput.
	• Scenario 2: Read-Heavy Workloads
	  • In workloads where most transactions are read-heavy and involve minimal writes, read-committed isolation can significantly reduce contention and improve overall system performance.
	• Scenario 3: Avoiding Write Lock Contention
	  • When multiple transactions need to read the same data while another transaction holds an exclusive lock (xlock) for write, read-committed allows the reading transactions to proceed without blocking, reducing contention for write locks.
	
	It's important to note that while read-committed isolation offers benefits in terms of concurrency, it may lead to non-repeatable reads, where the same read operation by a transaction yields different results if executed at different points in time. Application developers should consider the trade-offs between concurrency and data consistency when choosing an isolation level.

5.40.	The method **nextTransactionNumber** is synchronized in the **Transaction** class to ensure that transaction numbers are assigned sequentially and without conflicts when multiple threads concurrently request the next transaction number.

	Synchronization is necessary for this method because it involves reading and updating a shared counter, and without synchronization, race conditions could occur. On the other hand, other methods in the **Transaction** class may not require synchronization because they typically deal with operations that are inherently atomic to a single transaction. For example, when a transaction pins a block, locks a block, or reads/writes data, these operations are isolated to that specific transaction and do not involve shared data structures or counters that multiple transactions access simultaneously. In summary, synchronization is applied to **nextTransactionNumber** to ensure that transaction numbers are assigned in a thread-safe manner, while other methods are not synchronized because they deal with operations that are confined to the scope of an individual transaction and do not have shared state that requires protection against concurrent access.
5.41.a.	Yes, a transaction can pin a block without locking it. Pinning a block means loading it into the transaction's buffer pool, allowing the transaction to read and modify the contents of the block. Locking a block, on the other hand, typically involves acquiring a lock on the block to control concurrent access by other transactions. A transaction can choose to pin a block for its exclusive use without necessarily acquiring a lock, but it means that other transactions may still be able to acquire locks on the same block concurrently.
5.41.b.	Yes, a transaction can lock a block without pinning it. Locking a block means acquiring a lock on the block to control access by other transactions. It doesn't necessarily require the transaction to load the block into its buffer pool by pinning it. The transaction can request and obtain a lock on a block to prevent other transactions from accessing it, even if it doesn't plan to read or write the block immediately. This allows the transaction to control access to the block without incurring the overhead of loading it into memory.