Assignment 52-61 Sample Solution

7.13 Given the relation $R_1 = (A, B, C, D)$ the set of functional dependencies $F_1 = A \rightarrow B, C \rightarrow D, B \rightarrow C$ allows three distinct BCNF decompositions.

$$R_1 = \{(A, B), (C, D), (B, C)\}$$

is in BCNF as is

$$R_2 = \{(A, B), (C, D), (A, C)\}$$
$$R_2 = \{(A, B), (C, D), (A, C)\}$$
$$R_3 = \{(B, C), (A, D), (A, B)\}$$

7.14 Suppose $R$ is in 3NF according to the textbook definition. We show that it is in 3NF according to the definition in the exercise. Let $A$ be a nonprime attribute in $R$ that is transitively dependent on a key $\alpha$ for $R$. Then there exists $\beta \subseteq R$ such that $\beta \rightarrow A, \alpha \rightarrow \beta, A \not\in \alpha, A \not\in \beta$, and $\beta \rightarrow \alpha$ does not hold. But then $\beta \rightarrow A$ violates the textbook definition of 3NF since

- $A \not\in \beta$ implies $\beta \rightarrow A$ is nontrivial
- Since $\beta \rightarrow A$ does not hold, $\beta$ is not a superkey
- $A$ is not any candidate key, since $A$ is nonprime

Now, we show that if $R$ is in 3NF according to the exercise definition, it is in 3NF according to the textbook definition. Suppose $R$ is not in 3NF according to the textbook definition. Then there is an FD $\alpha \rightarrow \beta$ that fails all three conditions. Thus

- $\alpha \rightarrow \beta$ is nontrivial.
- $\alpha$ is not a superkey for $R$.
- Some $A$ in $\beta \rightarrow A$ is not in any candidate key.

This implies that $A$ is nonprime and $\alpha \rightarrow A$. Let $\gamma$ be a candidate key for $R$. Then $\gamma \rightarrow \alpha, \alpha \rightarrow \gamma$ does not hold (since $\alpha$ is not a superkey), $A \not\in \alpha$, and $A \not\in \gamma$ (since $A$ is nonprime). Thus $A$ is transitively dependent on $\gamma$, violating the exercise definition.

7.15 Referring to the definitions in Practice Exercise 7.14, a relation schema $R$ is said to be in 3NF if there is no non-prime attribute $A$ in $R$ for which $A$ is transitively dependent on a key for $R$. We can also rewrite the definition of 2NF given here as: “A relation schema $R$ is in 2NF if no non-prime attribute $A$ is partially dependent on any candidate key for $R.” To prove that every 3NF schema is in 2NF, it suffices to show that if a nonprime attribute $A$ is partially dependent on a candidate key $a$, then $A$ is also transitively dependent on the key $a$. Let $A$ be a non-prime attribute in $R$. Let $a$ be a candidate key for $R$. Suppose $A$ is partially dependent on $a$.

- From the definition of a partial dependency, we know that for some proper subset $\beta$ of $a, \beta \rightarrow A$.
- Since $\beta \subset a, a \rightarrow \beta$. Also, $\beta \rightarrow a$ does not hold, since $a$ is a candidate key.
- Finally, since $A$ is non-prime, it cannot be in either $\beta$ or $a$.

Thus we conclude that $a \rightarrow A$ is a transitive dependency. Hence, we have proved that every 3NF schema is also in 2NF.

7.16 The relation schema $R = (A, B, C, D, E)$ and the set of dependencies

$$A \rightarrow BC$$
$$B \rightarrow CD$$
$$E \rightarrow AD$$

constitute a BCNF decomposition, however it is clearly not in 4NF. (It is BCNF because all FDs are trivial).
7.18 Certain functional dependencies are called trivial functional dependencies because they are satisfied by all relations.

7.22 Computing \( B^+ \) by the algorithm in Figure 8.8 we start with \( \text{result} = \{B\} \). Considering FDs of the form \( b \rightarrow g \) in \( F \), we find that the only dependencies satisfying \( b \subseteq \text{result} \) are \( B \rightarrow B \) and \( B \rightarrow D \). Therefore \( \text{result} = \{B, D\} \). No more dependencies in \( F \) apply now. Therefore \( B^+ = \{B, D\} \).

7.23

Following the hint, use the following example of \( r \):

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_1 )</td>
<td>( b_1 )</td>
<td>( c_1 )</td>
<td>( d_1 )</td>
<td>( e_1 )</td>
</tr>
<tr>
<td>( a_2 )</td>
<td>( b_2 )</td>
<td>( c_1 )</td>
<td>( d_2 )</td>
<td>( e_2 )</td>
</tr>
</tbody>
</table>

With \( R_1 = \{A, B, C\} \), \( R_2 = \{C, D, E\} \):

a. \( \Pi_{R_1}(r) \) would be:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_1 )</td>
<td>( b_1 )</td>
<td>( e_1 )</td>
</tr>
<tr>
<td>( a_2 )</td>
<td>( b_2 )</td>
<td>( e_1 )</td>
</tr>
</tbody>
</table>

b. \( \Pi_{R_2}(r) \) would be:

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_1 )</td>
<td>( d_1 )</td>
<td>( e_1 )</td>
</tr>
<tr>
<td>( c_2 )</td>
<td>( d_2 )</td>
<td>( e_2 )</td>
</tr>
</tbody>
</table>

c. \( \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \) would be:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_1 )</td>
<td>( b_1 )</td>
<td>( c_1 )</td>
<td>( d_1 )</td>
<td>( e_1 )</td>
</tr>
<tr>
<td>( a_1 )</td>
<td>( b_1 )</td>
<td>( c_1 )</td>
<td>( d_2 )</td>
<td>( e_2 )</td>
</tr>
<tr>
<td>( a_2 )</td>
<td>( b_2 )</td>
<td>( c_1 )</td>
<td>( d_1 )</td>
<td>( e_1 )</td>
</tr>
<tr>
<td>( a_2 )</td>
<td>( b_2 )</td>
<td>( c_1 )</td>
<td>( d_2 )</td>
<td>( e_2 )</td>
</tr>
</tbody>
</table>

Clearly, \( \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \neq r \). Therefore, this is a lossy join.

7.26 BCNF is not always dependency preserving. Therefore, we may want to choose another normal form (specifically, 3NF) in order to make checking dependencies easier during updates. This would avoid joins to check dependencies and increase system performance.

7.29

\( A \rightarrow BC \) holds on the following table:

<table>
<thead>
<tr>
<th>( r ) :</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_1 )</td>
<td>( b_1 )</td>
<td>( c_1 )</td>
<td>( d_1 )</td>
<td></td>
</tr>
<tr>
<td>( a_1 )</td>
<td>( b_2 )</td>
<td>( c_2 )</td>
<td>( d_2 )</td>
<td></td>
</tr>
<tr>
<td>( a_1 )</td>
<td>( b_1 )</td>
<td>( c_1 )</td>
<td>( d_2 )</td>
<td></td>
</tr>
<tr>
<td>( a_1 )</td>
<td>( b_2 )</td>
<td>( c_2 )</td>
<td>( d_1 )</td>
<td></td>
</tr>
</tbody>
</table>

If \( A \rightarrow B \), then we know that there exists \( t_1 \) and \( t_2 \) such that \( t_1[B] = t_2[B] \). Thus, we must choose one of the following for \( t_1 \) and \( t_2 \):

- \( t_1 = r_1 \) and \( t_2 = r_3 \), or \( t_1 = r_3 \) and \( t_2 = r_1 \):
  - Choosing either \( t_2 = r_2 \) or \( t_2 = r_4 \), \( t_1[C] \neq t_2[C] \).
- \( t_1 = r_2 \) and \( t_2 = r_4 \), or \( t_1 = r_4 \) and \( t_2 = r_2 \):
  - Choosing either \( t_2 = r_1 \) or \( t_2 = r_3 \), \( t_1[C] \neq t_2[C] \).
Therefore, the condition $t_3[C] = t_2[C]$ cannot be satisfied, so the conjecture is false.

7.30
4NF is more desirable than BCNF because it reduces the repetition of information. If we consider a BCNF schema not in 4NF, we observe that decomposition into 4NF does not lose information provided that a lossless join decomposition is used, yet redundancy is reduced.

15.1 Even in this case the recovery manager is needed to perform roll-back of aborted transactions.

15.5 Most of the concurrency control protocols (protocols for ensuring that only serializable schedules are generated) used in practise are based on conflict serializability—they actually permit only a subset of conflict serializable schedules. The general form of view serializability is very expensive to test, and only a very restricted form of it is used for concurrency control.

15.6 There is a serializable schedule corresponding to the precedence graph below, since the graph is acyclic. A possible schedule is obtained by doing a topological sort, that is, $T_1, T_2, T_3, T_4, T_5$.

15.7 A cascadeless schedule is one where, for each pair of transactions $T_i$ and $T_j$ such that $T_j$ reads data items previously written by $T_i$, the commit operation of $T_i$ appears before the read operation of $T_j$. Cascadeless schedules are desirable because the failure of a transaction does not lead to the aborting of any other transaction. Of course this comes at the cost of less concurrency. If failures occur rarely, so that we can pay the price of cascading aborts for the increased concurrency, noncascadeless schedules might be desirable.

15.8 The ACID properties, and the need for each of them are:

* **Consistency**: Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database. This is typically the responsibility of the application programmer who codes the transactions.

* **Atomicity**: Either all operations of the transaction are reflected properly in the database, or none are. Clearly lack of atomicity will lead to inconsistency in the database.

* **Isolation**: When multiple transactions execute concurrently, it should be the case that, for every pair of transactions $T_i$ and $T_j$, it appears to $T_i$ that either $T_j$ finished execution before $T_i$ started, or $T_j$ started execution after $T_i$ finished. Thus, each transaction is unaware of other transactions executing concurrently with it. The user view of a transaction system requires the isolation property, and the property that concurrent schedules take the system from one consistent state to another. These requirements are satisfied by ensuring that only serializable schedules of individually consistency preserving transactions are allowed.

* **Durability**: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

15.9 The possible sequences of states are:

a. active ! partially committed ! committed. This is the normal sequence a successful transaction will follow. After executing all its statements it enters the partially committed state. After enough recovery information has been written to disk, the transaction finally enters the committed state.

b. active ! partially committed ! aborted. After executing the last statement of the transaction, it enters the partially committed state. But before enough recovery information is written to disk, a hardware failure may occur destroying the memory contents. In this case the changes which it made to the database are undone, and the transaction enters the aborted state.
c. **active** \( \rightarrow \) **failed** \( \rightarrow \) **aborted**. After the transaction starts, if it is discovered at some point that normal execution cannot continue (either due to internal program errors or external errors), it enters the failed state. It is then rolled back, after which it enters the **aborted** state.

15.10 A schedule in which all the instructions belonging to one single transaction appear together is called a **serial schedule**. A **serializable schedule** has a weaker restriction that it should be **equivalent** to some serial schedule. There are two definitions of schedule equivalence – conflict equivalence and view equivalence. Both of these are described in the chapter.

15.11

a. There are two possible executions: \( T_{13} T_{14} \) and \( T_{14} T_{13} \).

<table>
<thead>
<tr>
<th>Case 1:</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>initially</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>after ( T_{13} )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>after ( T_{14} )</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Consistency met: \( A = 0 \lor B = 0 = T \lor F = T \)

<table>
<thead>
<tr>
<th>Case 2:</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>initially</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>after ( T_{14} )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>after ( T_{13} )</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Consistency met: \( A = 0 \lor B = 0 = F \lor T = T \)

b. Any interleaving of \( T_{13} \) and \( T_{14} \) results in a non-serializable schedule.

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(B)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(A)</td>
</tr>
<tr>
<td>if ( A = 0 ) then ( B = B + 1 )</td>
<td>if ( B = 0 ) then ( A = A + 1 )</td>
</tr>
<tr>
<td>write(B)</td>
<td>write(A)</td>
</tr>
<tr>
<td>( T_{13} )</td>
<td>( T_{14} )</td>
</tr>
<tr>
<td>read(A)</td>
<td>read(B)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(A)</td>
</tr>
<tr>
<td>if ( A = 0 ) then ( B = B + 1 )</td>
<td>if ( B = 0 ) then ( A = A + 1 )</td>
</tr>
<tr>
<td>write(B)</td>
<td>write(A)</td>
</tr>
</tbody>
</table>

c. There is no parallel execution resulting in a serializable schedule. From part a. we know that a serializable schedule results in \( A = 0 \lor B = 0 \). Suppose we start with \( T_{13} \) **read(A)**. Then when the schedule ends, no matter when we run the steps of \( T_2 \), \( B = 1 \). Now suppose we start executing \( T_{14} \) prior to completion of \( T_{13} \). Then \( T_2 \) **read(B)** will give \( B \) a value of 0. So when \( T_2 \) completes, \( A = 1 \). Thus \( B = 1 \lor A = 1 \land \sim (A = 0 \land B = 0) \). Similarly for starting with \( T_{14} \) **read(B)**.
A recoverable schedule is one where, for each pair of transactions $T_i$ and $T_j$ such that $T_j$ reads data items previously written by $T_i$, the commit operation of $T_i$ appears before the commit operation of $T_j$. Recoverable schedules are desirable because failure of a transaction might otherwise bring the system into an irreversibly inconsistent state. Non-recoverable schedules may sometimes be needed when updates must be made visible early due to time constraints, even if they have not yet been committed, which may be required for very long duration transactions.

Transaction-processing systems usually allow multiple transactions to run concurrently. It is far easier to insist that transactions run serially. However there are two good reasons for allowing concurrency:

- Improved throughput and resource utilization. A transaction may involve I/O activity, CPU activity. The CPU and the disk in a computer system can operate in parallel. This can be exploited to run multiple transactions in parallel. For example, while a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU. This increases the throughput of the system.
- Reduced waiting time. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. It reduces the unpredictable delays and the average response time.

The recovery scheme using a log with deferred updates has the following advantages over the recovery scheme with immediate updates:

- The scheme is easier and simpler to implement since fewer operations and routines are needed, i.e., no UNDO.
- The scheme requires less overhead since no extra I/O operations need to be done until commit time (log records can be kept in memory the entire time).
- Since the old values of data do not have to be present in the log-records, this scheme requires less log storage space.

The disadvantages of the deferred modification scheme are:

- When a data item needs to accessed, the transaction can no longer directly read the correct page from the database buffer, because a previous write by the same transaction to the same data item may not have been propagated to the database yet. It might have updated a local copy of the data item and deferred the actual database modification. Therefore finding the correct version of a data item becomes more expensive.
- This scheme allows less concurrency than the recovery scheme with immediate updates. This is because write-locks are held by transactions till commit time.
- For long transaction with many updates, the memory space occupied by log records and local copies of data items may become too high.

The first phase of recovery is to undo the changes done by the failed transactions, so that all data items which have been modified by them get back the values they had before the first of the failed transactions started. If several of the failed transactions had modified the same data item, forward processing of log-records for undo-list transactions would make the data item get the value which it had before the last failed transaction to modify that data item started. This is clearly wrong, and we can see that reverse processing gets us the desired result.

The second phase of recovery is to redo the changes done by committed transactions, so that all data items which have been modified by them are restored to the value they had after the last of the committed transactions finished. It can be seen that only forward processing of log-records belonging to redo-list transactions can guarantee this.
Consider the bank account $A$ with balance $100$. Consider two transactions $T_1$ and $T_2$ each depositing $10$ in the account. Thus the balance would be $120$ after both these transactions are executed. Let the transactions execute in sequence: $T_1$ first and then $T_2$. The log records corresponding to the updates of $A$ by transactions $T_1$ and $T_2$ would be $<T_1, A, 100, 110>$ and $<T_2, A, 110, 120>$ resp.

Say, we wish to undo transaction $T_1$. The normal transaction undo mechanism will replaces the value in question—$A$ in this example—by the old value field in the log record. Thus if we undo transaction $T_1$ using the normal transaction undo mechanism the resulting balance would be $100$ and we would, in effect, undo both transactions, whereas we intend to undo only transaction $T_1$.

Let the erroneous transaction be $T_e$.

1. Identify the latest checkpoint, say $C$, in the log before the log record $<T_e, START>$.
2. Redo all log records starting from the checkpoint $C$ till the log record $<T_e, COMMIT>$. Some transaction—apart from transaction $T_e$—would be active at the commit time of transaction $T_e$. Let $S_1$ be the set of such transactions.
3. Rollback $T_e$ and the transactions in the set $S_1$.
4. Scan the log further starting from the log record $<T_e, COMMIT>$ till the end of the log. Note the transactions that were started after the commit point of $T_e$. Let the set of such transactions be $S_2$. Re-execute the transactions in set $S_1$ and $S_2$ logically.

17.6

We can maintain the LSNs of such pages in an array in a separate disk page. The LSN entry of a page on the disk is the sequence number of the latest log record reflected on the disk. In the normal case, as the LSN of a page resides in the page itself, the page and its LSN are in consistent state. But in the modified scheme as the LSN of a page resides in a separate page it may not be written to the disk at a time when the actual page is written and thus the two may not be in consistent state.

If a page is written to the disk before its LSN is updated on the disk and the system crashes then, during recovery, the page LSN read from the LSN array from the disk is older than the sequence number of the log record reflected to the disk. Thus some updates on the page will be redone unnecessarily but this is fine as updates are idempotent. But if the page LSN is written to the disk to before the actual page is written and the system crashes then some of the updates to the page may be lost. The sequence number of the log record corresponding to the latest update to the page that made to the disk is older than the page LSN in the LSN array and all updates to the page between the two LSNs are lost.

Thus the LSN of a page should be written to the disk only after the page has been written and; we can ensure this as follows: before writing a page containing the LSN array to the disk, we should flush the corresponding pages to the disk. (We can maintain the page LSN at the time of the last flush of each page in the buffer separately, and avoid flushing pages that have been flushed already.)

17.7

Volatile storage is storage which fails when there is a power failure. Cache, main memory, and registers are examples of volatile storage. Non-volatile storage is storage which retains its content despite power failures. An example is magnetic disk. Stable storage is storage which theoretically survives any kind of failure (short of a complete disaster!). This type of storage can only be approximated by replicating data.

In terms of I/O cost, volatile memory is the fastest and non-volatile storage is typically several times slower. Stable storage is slower than non-volatile storage because of the cost of data replication.

17.8

a. Stable storage cannot really be implemented because all storage devices are made of hardware, and all hardware is vulnerable to mechanical or electronic device failures.
b. Database systems approximate stable storage by writing data to multiple storage devices simultaneously. Even if one of the devices crashes, the data will still be available on a different device. Thus data loss becomes extremely unlikely.

17.11
Consider a banking scheme and a transaction which transfers $50 from account $A$ to account $B$. The transaction has the following steps:

a. $\text{read}(A,a_1)$

b. $a_1 := a_1 - 50$

c. $\text{write}(A,a_1)$

d. $\text{read}(B,b_1)$

e. $b_1 := b_1 + 50$

f. $\text{write}(B,b_1)$

Suppose the system crashes after the transaction commits, but before its log records are flushed to stable storage. Further assume that at the time of the crash the update of $A$ in the third step alone had actually been propagated to disk whereas the buffer page containing $B$ was not yet written to disk. When the system comes up it is in an inconsistent state, but recovery is not possible because there are no log records corresponding to this transaction in stable storage.

17.14
a. Two very safe is suitable here because it guarantees durability of updates by committed transactions, though it can proceed only if both primary and backup sites are up. Availability is low, but it is mentioned that this is acceptable.

b. One safe committing is fast as it does not have to wait for the logs to reach the backup site. Since data loss can be tolerated, this is the best option.

c. With two safe committing, the probability of data loss is quite low, and also commits can proceed as long as at least the primary site is up. Thus availability is high. Commits take more time than in the one safe protocol, but that is mentioned as acceptable.

1. The read-committed isolation level ensures that a transaction reads only the committed data. A transaction $T_i$ cannot read a data item $X$ which has been modified by a yet uncommitted concurrent transaction $T_j$. This makes $T_i$ independent of the success or failure of $T_j$. Hence, the schedules which follow read committed isolation level become cascade free.

2. 

a. Read Uncommitted:

```
\begin{array}{c|c|c|c}
T_1 & & T_2 \\
\text{read}(A) & \text{write}(A) & \text{read}(A) & \text{write}(A) \\
\text{write}(A) & & \text{read}(A) & \\
\end{array}
```

In the above schedule, $T_2$ reads the value of $A$ written by $T_1$ even before $T_1$ commits. This schedule is not serializable since $T_1$ also reads a value written by $T_2$, resulting in a cycle in the precedence graph.
b. Read Committed:

In the above schedule, the first time \( T_1 \) reads \( A \), it sees a value of \( A \) before it was written by \( T_2 \), while the second read \( (A) \) by \( T_1 \) sees the value written by \( T_2 \) (which has already committed). The first read results in \( T_1 \) preceding \( T_2 \), while the second read results in \( T_2 \) preceding \( T_1 \), and thus the schedule is not serializable.

c. Repeatable Read:

Consider the following schedule, where \( T_1 \) reads all tuples in \( r \) satisfying predicate \( P \); to satisfy repeatable read, it must also share-lock these tuples in a two-phase manner.

Suppose that the tuple \( t \) inserted by \( T_2 \) satisfies \( P \); then the insert by \( T_2 \) causes \( T_2 \) to serialized after \( T_1 \), since \( T_1 \) does not see \( t \). However, the final read \( (A) \) operation of \( T_1 \) forces \( T_2 \) to precede \( T_1 \), causing a cycle in the precedence graph.

3.

a. The repeatable read schedule in the preceding question is an example of a schedule exhibiting the phantom phenomenon and is non-serializable.

b. Consider the schedule

Suppose that tuple \( t \) deleted by \( T_2 \) is from relation \( r \), but does not satisfy predicate \( P \), for example because its \( A \) value is 3. Then, there is no phantom conflict between \( T_1 \) and \( T_2 \), and \( T_2 \) can be serialized before \( T_1 \).