UNIVERSITY OF CHITTAGONG

Department of Computer Science & Engineering

Program: **B.Sc.** (Engineering)
Session: 2022-2023
4th Semester

Class Assignment
Topic: Practice Exercises

Course Title: DataBase Systems
Course Code: CSE - 413

Submitted To:
Dr. Rudra Pratap Deb Nath
Associate Professor
Department of Computer Science & Engineering
University of Chittagong

Submitted By:
Nilanjana Das Jui
ID: 23701011
Dept. of Computer Science & Engineering

Date of submission: July 07, 2025

Contents

Chapter - 07: Relational Database Design

Question - 7.8:

Consider the algorithm in Figure 7.19 to compute α^+. Show that this algorithm is more efficient than the one presented in Figure 7.8 (Section 7.4.2) and that it computes α^+ correctly.

Answer:

```
result := α;
repeat
    for each functional dependency β → γ in F do
        begin
            if β ⊆ result then result := result ∪ γ;
        end
until (result does not change)
```

Figure 7.8 An algorithm to compute α^+, the closure of α under F.

Explanation:

- Begin with result = α.
- Repeatedly scan all FDs.
- If the left-hand side (β) of a dependency is a subset of the current result, add the right-hand side (γ) to the result.
- Repeat until the result stops changing.

Time Complexity:

Worst case: $\mathcal{O}(m \times n^2)$, where $m = |F|$ (number of functional dependencies), and n is the number of attributes.

This is because:

- In each iteration, we scan all FDs.
- Each iteration may update result.
- We may require up to n iterations in the worst case if one new attribute is added at a time.

The algorithm in Figure 7.19 to compute α^+:

```
result := ∅;
/* fdcount is an array whose ith element contains the number
   of attributes on the left side of the ith FD that are
   not yet known to be in α+ */
for i := 1 to |F| do
   begin
     let β → γ denote the ith FD;
     fdcount [i] := |β|;
   end
/* appears is an array with one entry for each attribute. The
   entry for attribute A is a list of integers. Each integer
   i on the list indicates that A appears on the left side
   of the ith FD */
for each attribute A do
   begin
     appears [A] := NIL;
     for i := 1 to |F| do
       begin
         let β → γ denote the ith FD;
         if A ∈ β then add i to appears [A];
       end
   end
addin (α);
return (result);

procedure addin (α);
for each attribute A in α do
   begin
     if A ∉ result then
       begin
         result := result ∪ {A};
         for each element i of appears[A] do
           begin
             fdcount [i] := fdcount [i] − 1;
             if fdcount [i] := 0 then
               begin
                 let β → γ denote the ith FD;
                 addin (γ);
               end
           end
       end
   end
```

Figure 7.19 An algorithm to compute α^+.

Explanation

- The algorithm starts by scanning all functional dependencies in F. For each functional dependency $\beta \to \gamma$, it counts the number of attributes in the left-hand side β. This count is stored in `fdcount[i]` for each i^{th} FD. This tells us how many more attributes need to be added to the closure before this FD can be triggered.

- Next, for every attribute A in the schema, the algorithm constructs a list `appears[A]` which contains the indices of all FDs where A appears on the left-hand side (β). This mapping helps us later in efficiently updating the status of dependent FDs when A is added to the closure.
- The recursive procedure `addin(`α`)` begins with the input set α.
  - For each attribute A in α:
    * If A is not already in `result`, it is added.
    * For each FD index i in `appears[A]`, we decrement `fdcount[i]`.
    * If `fdcount[i]` becomes zero, this means all attributes in the left-hand side β of the i^{th} FD are now present in the closure.
    * Thus, the right-hand side γ of the FD is added by calling `addin(`γ`)`.
- This process continues recursively until no new attributes can be added. The recursion guarantees that each FD is used at most once — exactly when all attributes of its left-hand side are included in `result`.
- Once all recursive calls are complete, the algorithm returns `result`, which now contains the full closure α^+ under the given set of functional dependencies.

Time Complexity Analysis

- Let n be the number of attributes and $m = |F|$ be the number of functional dependencies.
- Initializing `fdcount[i]` for all FDs: $\mathcal{O}(m \cdot n)$ in the worst case (if each FD has up to n attributes).
- Building `appears[A]` requires checking every attribute in every FD: also $\mathcal{O}(m \cdot n)$.
- Each attribute is added to the result at most once: $\mathcal{O}(n)$.
- Each FD is triggered only once when its LHS becomes a subset of the result: $\mathcal{O}(m)$.
- Each appearance in `appears[A]` is processed once: Let k be the total number of these appearances.

$$\mathcal{O}(m \cdot n) \text{ (for initialization)} + \mathcal{O}(n + m + k) = \mathcal{O}(m \cdot n + k)$$

Since $k \leq m \cdot n$, the total complexity is effectively:

$$\boxed{\mathcal{O}(m \cdot n)}$$

This is significantly better than the naive algorithm, which may take $\mathcal{O}(m \cdot n^2)$ due to repeated scanning.

Efficiency Comparison with Figure 7.8

The algorithm in Figure 7.8 uses a simpler approach:

```
result := α;
repeat
   for each functional dependency β → γ in F do
      if β ⊆ result then result := result ∪ γ
until result does not change
```

This approach repeatedly scans the entire list of FDs until no new attributes can be added to the closure. In the worst case, it may take up to n iterations (where n is the number of attributes), and in each iteration, it checks all FDs.

Time Complexity

- **Figure 7.8:** $\mathcal{O}(m \cdot n^2)$ in the worst case, where $m = |F|$ is the number of FDs and n is the number of attributes. This is because each FD is scanned multiple times, and each scan can take $\mathcal{O}(n)$ time.
- **Figure 7.19:** $\mathcal{O}(m + n + k)$ where k is the total number of FD appearances in `appears[A]`. Each FD is processed at most once when its left-hand side is satisfied, and no unnecessary iterations are performed.

So, Both algorithms compute α^+ correctly. However, the algorithm in Figure 7.19 is more efficient because:

- It avoids repeated scanning of the functional dependency set.
- Each FD is only triggered when all its left-hand attributes are known.
- The use of auxiliary data structures allows faster lookups and minimal recomputation.

Therefore, the Figure 7.19 algorithm is both correct and more efficient than the one in Figure 7.8.

Question - 7.26:

Consider the following proposed rule for functional dependencies: If $\alpha \to \beta$ and $\alpha \to \beta$,then $\alpha \to \gamma$.Prove that this rule is not sound by showing a relation r that satisfies $\alpha \to \beta$ and $\beta \to \gamma$,but does not satisfy $\alpha \to \gamma$.

Answer:

We will prove that the above rule is **not sound** by providing a counterexample.

Let the relation schema be:

$$R(\alpha, \gamma, \beta)$$

Define a relation r over R as follows:

α	γ	β
3	8	10
5	2	4
5	7	4

1. Check whether $\alpha \to \beta$ holds:

We examine if the same value of α always leads to the same β:

- When $\alpha = 3$, $\beta = 10$
- When $\alpha = 5$, both tuples have $\beta = 4$

Since every value of α determines a unique β, we conclude that:

$$\alpha \to \beta \text{ holds.}$$

2. Check whether $\gamma \to \beta$ holds:

- $\gamma = 8 \Rightarrow \beta = 10$
- $\gamma = 2 \Rightarrow \beta = 4$
- $\gamma = 7 \Rightarrow \beta = 4$

Each value of γ is associated with a unique β, so:

$$\gamma \to \beta \text{ holds.}$$

3. Check whether $\alpha \to \gamma$ holds:

- For $\alpha = 3$, $\gamma = 8$
- For $\alpha = 5$, $\gamma = 2$ and $\gamma = 7$

Since $\alpha = 5$ is associated with multiple γ values, the dependency does not hold:

$$\alpha \to \gamma \text{ does not hold.}$$

The relation r satisfies:

$$\alpha \to \beta \quad \text{and} \quad \gamma \to \beta$$

but does **not** satisfy:

$$\alpha \to \gamma$$

Therefore, the proposed inference rule:

$$\text{If } \alpha \to \beta \text{ and } \gamma \to \beta, \text{ then } \alpha \to \gamma$$

is **not sound**.

Question - 7.35:

Although the BCNF algorithm ensures that the resulting decomposition is lossless, it is possible to have a schema and a decomposition that was not generated by the algorithm, that is in BCNF, and is not lossless. Give an example of such a schema and its decomposition.

Answer:

Although the BCNF decomposition algorithm guarantees that any decomposition it produces will be **lossless**, it is important to note that there exist decompositions which are **in BCNF but not lossless** if they are not generated by the BCNF algorithm.

Consider the relation schema:

$$R = (A, B, C, D, E)$$

with the set of functional dependencies:

$$F = \{A \to BC, \quad CD \to E, \quad B \to D, \quad E \to A\}$$

We consider the decomposition of R into two relations:

$$R_1 = (A, B, C) \quad \text{and} \quad R_2 = (A, D, E)$$

Checking BCNF:

- For R_1, the FD $A \to BC$ holds, so A functionally determines all other attributes in R_1. Therefore, A is a key for R_1, and R_1 is in BCNF.
- For R_2, the only relevant FD is $E \to A$ (since $B \to D$ does not apply as $B \notin R_2$). Because E determines A and together these cover all attributes of R_2, E is a key for R_2, so R_2 is in BCNF as well.

Checking for Lossless Join:

The common attribute between R_1 and R_2 is:

$$R_1 \cap R_2 = \{A\}$$

The decomposition is lossless if either

$$A \to R_1 \quad \text{or} \quad A \to R_2$$

Since

$$A \to BC \quad (\text{which is } R_1 - \{A\}),$$

the condition $A \to R_1$ holds, so this decomposition should be lossless.

Counterexample with Data Showing Lossiness:

Consider the following instance of relation R:

A	B	C	D	E
5	9	12	14	18
6	10	12	15	19

Projecting this data onto $R_1 = (A, B, C)$:

A	B	C
5	9	12
6	10	12

Projecting onto $R_2 = (A, D, E)$:

A	D	E
5	14	18
6	15	19

Now, performing the natural join $R_1 \bowtie R_2$ on attribute A:

A	B	C	D	E
5	9	12	14	18
5	9	12	15	19
6	10	12	14	18
6	10	12	15	19

Observe that tuples such as $(5, 9, 12, 15, 19)$ and $(6, 10, 12, 14, 18)$ are *not* in the original relation R. These **spurious tuples** appear due to the join combining tuples with matching A values but mismatched D and E or B and C values.

Therefore, both R_1 and R_2 are in BCNF and the theoretical condition for losslessness is satisfied, this decomposition is actually **lossy**.