**7.1 Suppose that we decompose the schema R = (A, B, C, D, E) into (A, B, C) (A, D, E). Show that this decomposition is a lossless decomposition if the following set F of functional dependencies holds:**

**A → BC**

**CD → E**

**B → D**

**E → A**

To show that the decomposition of schema R=(A,B,C,D,E) into two relations R1=(A,B,C) and R2=(A,D,E) is lossless, we need to verify if the lossless join condition holds.

The lossless decomposition criterion states that the decomposition of R into two relations R1​ and R2​ is lossless if at least one of the following conditions holds:

1. The intersection of the attribute sets of R1​ and R2​ (i.e., R1∩R2​) is a **key** for at least one of the relations.

Now, let's break down the given functional dependencies:

**Given:**

1. A→BC (This means that knowing A determines both B and C.)
2. CD→E (This means that knowing C and D together determines E.)
3. B→D (This means that knowing B determines D.)
4. E→A (This means that knowing E determines A.)

**Step 1: Identify the intersection of R1​ and R2​**

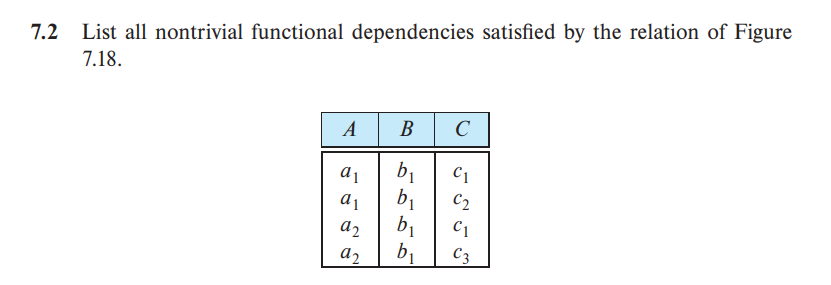
* R1=(A,B,C)
* R2=(A,D,E)
* The intersection of R1​ and R2 is A.

**Step 2: Determine if A is a key for any of the relations**

* For R1=(A,B,C), we need to check if A is a key.
  + From A→BC, knowing A gives us B and C. Therefore, A determines all the attributes in R1​, meaning A is a key for R1​.
* For R2=(A,D,E), we need to check if A is a key.
  + From the given functional dependencies, A→BC, and B→D, we can derive A→D because A→B and B→D.
  + So, A determines B, D, and C in R2​, meaning A determines all attributes in R2​. Therefore, A is a key for R2​.

**Step 3: Conclusion**

Since the intersection of R1​ and R2​ is A, and Ais a key for both R1​ and R2​, the decomposition of R into R1​ and R2​ is **lossless**.

****

To find the nontrivial functional dependencies satisfied by the relation R=(A,B,C), we need to understand what functional dependencies can exist between the attributes A, B, and C based on the given values for these attributes.

provided values:

* A={a1,a1,a2,a2}
* B={b1,b1,b1,b1}
* C={c1,c2,c1,c3}

We can analyze the possible functional dependencies by looking at how attributes in the relation are related to each other. A functional dependency X→Y means that if two tuples have the same values for attributes in X, they must also have the same values for attributes in Y.

**Step 1: Look at A→B and A→C**

* In the given relation, we can see that whenever the values of A are the same (i.e., a1 and a2), the corresponding values of B and C are the same.
* For example:
  + When A=a1, B=b1 and C=c1 or c2.
  + When A=a2, B=b1 and C=c1 or c3.

However, since B is always b1, it is trivially determined by any value of A. So, we can say A→B is a functional dependency.

* For A→C, note that different values of C appear for the same values of A. So, A does not determine C, and thus A→C is **not** a functional dependency.

**Step 2: Look at B→A and B→C**

* Since B always has the same value (i.e., b1) for all tuples, it does not provide any new information about the values of A or C. Thus, we cannot say B→A or B→C holds as functional dependencies.

**Step 3: Look at C→A and C→B**

* For C→A, we see that when C=c1, A can either be a1 or a2, so C does not uniquely determine A.
* For C→B, since B=b1 always, C trivially determines B. Therefore, C→B is a valid functional dependency.

**Step 4: List the nontrivial functional dependencies**

From the analysis, the nontrivial functional dependencies are:

1. A→B (since for each value of A, B is always b1)
2. C→B (since for each value of C, B is always b1)

These are the only nontrivial functional dependencies in the relation.

**7.3 Explain how functional dependencies can be used to indicate the following:**

**• A one-to-one relationship set exists between entity sets student and instructor.**

**• A many-to-one relationship set exists between entity sets student and instructor**

Functional dependencies (FDs) can help express relationships between entity sets in a database schema by showing how one attribute (or a set of attributes) determines another. In the context of the relationship between entity sets like **Student** and **Instructor**, FDs can be used to indicate one-to-one and many-to-one relationships.

**1. One-to-One Relationship between Student and Instructor**

A **one-to-one** relationship means that each student is associated with exactly one instructor, and each instructor is responsible for exactly one student. In terms of functional dependencies, this can be represented by the following:

* StudentID→InstructorID
  + This means that each student is assigned exactly one instructor, so knowing a student’s ID gives you the instructor’s ID.
  + **Symmetric Dependency**: You could also have InstructorID→StudentID, meaning that each instructor is associated with exactly one student, so knowing the instructor’s ID gives you the student’s ID.

In this case, the relationship is **bijective**, meaning each student has a unique instructor, and each instructor is assigned to exactly one student. The functional dependency StudentID→InstructorID (and its reverse) accurately reflects this one-to-one relationship.

**2. Many-to-One Relationship between Student and Instructor**

A **many-to-one** relationship means that multiple students can be associated with the same instructor, but each student is assigned to exactly one instructor. In terms of functional dependencies, this can be represented as:

* StudentID→InstructorID
  + This means that each student is assigned exactly one instructor. Knowing a student’s ID determines the instructor’s ID.
  + The reverse, InstructorID→StudentID, does **not** hold in this case because many students can share the same instructor.

Here, multiple students may have the same instructor, but each student is associated with exactly one instructor. The functional dependency StudentID→InstructorID captures the many-to-one relationship, as one instructor may have many students, but each student has one specific instructor.

**Summary:**

* **One-to-One**: Both StudentID→InstructorID and InstructorID→StudentID are true.
* **Many-to-One**: Only StudentID→InstructorID holds, but not the reverse.

**7.4 Use Armstrong’s axioms to prove the soundness of the union rule. (Hint: Use the augmentation rule to show that, if α → β, then α → αβ. Apply the augmentation rule again, using α → γ, and then apply the transitivity rule.)**

Armstrong’s Axioms are a set of inference rules used to derive functional dependencies (FDs) in a relational database schema. The **union rule** states that if we have two functional dependencies α→β and α→γ, we can infer that:

α→βγ

In other words, if α determines both β and γ, then α determines the union of β and γ.

**Proof of Soundness of the Union Rule using Armstrong’s Axioms**

To prove this using Armstrong’s Axioms, we can follow these steps:

1. **Start with the given functional dependencies**:
   * α→β
   * α→γ
2. **Apply the Augmentation Rule**:
   * The **augmentation rule** states that if α→β, then for any set of attributes δ, αδ→βδ.
   * Here, let’s augment both β and γ by γ (which is part of the second FD). Applying the augmentation rule on α→β, we get:

α→βγ

This follows because we augmented both sides of the functional dependency α→β with γ.

1. **Now, apply the Transitivity Rule**:
   * The **transitivity rule** says that if α→β and β→γ, then α→γ.
   * In this case, we don’t directly apply transitivity to the augmented dependency. However, we already have α→βγ, and we can conclude that α determines β and γ together.
2. **Conclusion**:
   * We have shown that α→β and α→γ together imply that α→βγ, which is exactly the statement of the union rule.

Thus, the union rule is **sound** according to Armstrong’s Axioms.

**7.5 Use Armstrong’s axioms to prove the soundness of the pseudo-transitivity rule.**

The **pseudotransitivity rule** is one of Armstrong's axioms and is formally expressed as follows:

If α→β and αγ→δ, then βγ→δ.

In words, if α→β (i.e., α determines β) and αγ→δ (i.e., α together with γ determines δ), then we can conclude that βγ determines δ (i.e., β together with γ determines δ).

**Proof of Soundness of the Pseudotransitivity Rule using Armstrong’s Axioms:**

We will prove the soundness of the pseudotransitivity rule using Armstrong’s axioms, particularly **augmentation** and **transitivity**.

**Step-by-step proof:**

**Given:**

* α→β (i.e., α determines β).
* αγ→δ (i.e., α together with γ determines δ).

We need to prove:

* βγ→δ (i.e., β together with γ determines δ).

**Step 1: Augment α→β with γ**

From the **augmentation rule**, which states that if α→β, then for any set of attributes γ, we have αγ→βγ, we can augment the given FD α→β with γ:

αγ→βγ

This is the result of applying the augmentation rule.

**Step 2: Use the second given dependency αγ→δ**

We are also given the second functional dependency αγ→δ. So we have:

αγ→δ

**Step 3: Apply the transitivity rule**

Now, using the **transitivity rule** of Armstrong’s axioms, which states that if αγ→βγ and αγ→δ, then we can infer βγ→δ, we can combine the two dependencies we have:

1. αγ→βγ (from augmentation).
2. αγ→δ (given).

By transitivity, we can infer:

βγ→δ

**Conclusion:**

We have shown that if α→β and αγ→δ, then βγ→δ, which is exactly the statement of the **pseudotransitivity rule**.

Thus, the pseudotransitivity rule is **sound** according to Armstrong’s axioms.

**7.6 Compute the closure of the following set F of functional dependencies for relation schema R = (A, B, C, D, E).**

**A → BC**

**CD → E**

**B → D**

**E → A**

**List the candidate keys for R.**

To compute the **closure** of a set of functional dependencies F and find the **candidate keys** for a relation schema R=(A,B,C,D,E), we need to follow these steps:

**Step 1: List the functional dependencies in F**

Given the functional dependencies:

* A→BC
* CD→E
* B→D
* E→A

**Step 2: Compute the closure of attributes**

We will use these functional dependencies to compute the closure of different sets of attributes and determine which sets can be candidate keys.

To find the closure of a set of attributes, we apply the given functional dependencies iteratively until no new attributes can be added to the closure.

**Start with the closure of individual attributes and combinations:**

1. **A+** (closure of A):
   * A→BC, so A+={A,B,C}.
   * B→D, so A+={A,B,C,D}.
   * CD→E, and we already have C and D, so A+={A,B,C,D,E}.
   * We have all attributes in R={A,B,C,D,E}, so A+={A,B,C,D,E}.

**Conclusion**: A+={A,B,C,D,E}, so A is a **candidate key**.

1. **B+** (closure of B):
   * B→D, so B+={B,D}.
   * We don't have enough to determine A, C, or E from B+, so we cannot form all attributes. Therefore, B+≠R, and B is **not a candidate key**.
2. **C+** (closure of C):
   * C+={C}.
   * No additional attributes can be derived from C, so C+≠R, and C is **not a candidate key**.
3. **D+** (closure of D):
   * D+={D}.
   * No additional attributes can be derived from D, so D+≠R, and D is **not a candidate key**.
4. **E+** (closure of E):
   * E→A, so E+={E,A}.
   * A→BC, so E+={E,A,B,C}.
   * B→D, so E+={E,A,B,C,D}.
   * CD→E, but we already have all attributes, so E+={A,B,C,D,E}.

**Conclusion**: E+={A,B,C,D,E}, so E is a **candidate key**.

**Step 3: Identify candidate keys**

* We found that both A and E determine all attributes of R, and thus both are candidate keys.
* Therefore, the **candidate keys** for R are:
  + {A}
  + {E}

**Final Answer:**

* The **closure** of the set F of functional dependencies results in the candidate keys A and E.

**7.7 Using the functional dependencies of Exercise 7.6, compute the canonical cover Fc.**

**A → BC**

**CD → E**

**B → D**

**E → A**

To compute the **canonical cover** (or **minimal cover**) Fc​ for a given set of functional dependencies F, we need to follow a series of steps to ensure that:

1. Each functional dependency has a single attribute on the right-hand side.
2. The set of functional dependencies is minimal (i.e., no redundant functional dependencies or attributes).

**Functional Dependencies Given:**

1. A→BC
2. CD→E
3. B→D
4. E→A

**Step 1: Make right-hand sides of all FDs have a single attribute**

First, we check each functional dependency to make sure the right-hand side contains only one attribute. If the right-hand side contains multiple attributes, we break it down.

* A→BC is a functional dependency with two attributes on the right-hand side. We break it into two separate FDs:
  + A→B
  + A→C

Now, the set of functional dependencies becomes:

1. A→B
2. A→C
3. CD→E
4. B→D
5. E→A

**Step 2: Remove redundant functional dependencies**

Now, we need to check if any of the functional dependencies are redundant (i.e., can be derived from others). We do this by computing the closure of attributes and checking if any FD can be inferred from the others.

* For each functional dependency, we check if it can be derived from the others by computing the closure of the left-hand side of the FD, excluding that particular FD.

**Checking A→B:**

* Compute the closure of A+ without considering A→B.
  + A+={A}
  + From A→C, we add C, so A+={A,C}.
  + Since we cannot derive B from the other dependencies, A→B is **not redundant**.

**Checking A→C:**

* Compute the closure of A+ without considering A→C.
  + A+={A}
  + From A→B, we add B, so A+={A,B}.
  + We cannot derive C from the other dependencies, so A→C is **not redundant**.

**Checking CD→E:**

* Compute the closure of CD+ without considering CD→E.
  + CD+={C,D}
  + From B→D, we get D, but it doesn't give us E. Thus, we cannot derive E without CD→E, so CD→E is **not redundant**.

**Checking B→D:**

* Compute the closure of B+ without considering B→D.
  + B+={B}
  + We cannot derive D from any other dependencies, so B→D is **not redundant**.

**Checking E→A:**

* Compute the closure of E+ without considering E→A.
  + E+={E}
  + From E→A, we get A, and then we can derive B and C from A→B and A→C. Thus, E→A can be derived from the other dependencies.
  + So, **E→A is redundant** and can be removed.

**Step 3: Resulting Canonical Cover**

After removing the redundant functional dependency E→A, the canonical cover Fc​ for the set F is:

1. A→B
2. A→C
3. CD→E
4. B→D

Thus, the **canonical cover** Fc​ is:

Fc​={A→B,A→C,CD→E,B→D}

**7.9 Given the database schema R(A, B, C), and a relation r on the schema R, write an SQL query to test whether the functional dependency B → C holds on relation r. Also write an SQL assertion that enforces the functional dependency. Assume that no null values are present. (Although part of the SQL standard, such assertions are not supported by any database implementation currently.)**

To test whether the functional dependency B→C holds on a relation r for the schema R(A,B,C), you can write an SQL query to check whether there are any instances where the same value of B maps to more than one value of C. If such a situation exists, the functional dependency B→C does **not** hold.

**SQL Query to Test B→C**

To test B→C, the query will check if there are any duplicate pairs of B and C where a single value of B corresponds to multiple values of C. This can be done using a GROUP BY and HAVING clause to find cases where B has more than one distinct value of C.

SELECT B

FROM r

GROUP BY B

HAVING COUNT(DISTINCT C) > 1;

**Explanation:**

* The query groups the rows by the value of B.
* It then checks if there are more than one distinct value of C for each value of B using the HAVING COUNT(DISTINCT C) > 1.
* If the query returns any rows, it means that for some values of B, there are multiple corresponding values of C, and therefore, the functional dependency B→C does **not** hold.

If the query returns an empty result, the functional dependency B→C **does** hold because every value of B maps to only one value of C.

**SQL Assertion to Enforce B→C**

An **SQL assertion** is used to enforce a constraint on a database. The assertion ensures that for every row in the table, if there are multiple rows with the same value of B, they must all have the same value of C. Although SQL assertions are part of the SQL standard, as noted, most database systems do not support them directly.

The assertion can be written as follows:

CREATE ASSERTION B\_TO\_C\_DEPENDENCY

CHECK (

NOT EXISTS (

SELECT 1

FROM r AS r1, r AS r2

WHERE r1.B = r2.B AND r1.C <> r2.C

)

);

**Explanation:**

* This assertion checks that for every pair of rows in the table r, if two rows have the same value of B (i.e., r1.B=r2.B), they must have the same value of C (i.e., r1.C=r2.C).
* If there exists a pair of rows where the values of B are the same, but the values of C are different, the assertion will fail, enforcing that B→C holds.

However, as mentioned earlier, many relational database management systems (RDBMS) do not support the CREATE ASSERTION syntax. In those cases, you might need to use triggers or other constraints to enforce functional dependencies manually.

**7.10 Our discussion of lossless decomposition implicitly assumed that attributes on the left-hand side of a functional dependency cannot take on null values. What could go wrong on decomposition, if this property is violated?**

If attributes on the **left-hand side** of a functional dependency **can take on null values**, the process of **lossless decomposition** can fail. This can cause issues such as **spurious tuples** appearing when we attempt to reconstruct the original relation by joining the decomposed tables.

**Problems When Left-Hand Side Attributes Can Be Null**

1. **Functional Dependency May Not Hold for Null Values**
   * In relational databases, **functional dependencies (FDs) do not apply to tuples where the left-hand side contains null values**.
   * For example, if we have A→B, it means that if two rows have the same value for A, they must also have the same value for B. However, if A is null in some rows, SQL does not enforce that B must be the same, since **null values are treated as unknown**.
2. **Spurious Tuples Due to Nulls in Decomposed Relations**
   * If a decomposition of relation R into R1​ and R2​ is performed, and the join condition involves an attribute that **can be null**, the natural join may produce **extra tuples (spurious tuples)** or **lose tuples**, making it impossible to reconstruct the original relation.
   * This happens because **joining on a null value results in missing matches**, meaning some original data might not be recovered.

**Example Scenario**

**Given Relation:**

Let’s assume we have a relation R(A,B,C) with the functional dependency:

A→B

and we decompose it into:

* R1​(A,B)
* R2​(A,C)

Now, consider the following data in R:

| **A** | **B** | **C** |
| --- | --- | --- |
| 1 | X | Y |
| 2 | Z | W |
| ∅ | P | Q |

After decomposition:

**Table R1​(A,B)**

| **A** | **B** |
| --- | --- |
| 1 | X |
| 2 | Z |
| ∅ | P |

**Table R2​(A,C)**

| **A** | **C** |
| --- | --- |
| 1 | Y |
| 2 | W |
| ∅ | Q |

When we try to join R1​ and R2​ using a **natural join on A**:

SELECT R1.A, R1.B, R2.C

FROM R1 NATURAL JOIN R2;

Expected output:

| **A** | **B** | **C** |
| --- | --- | --- |
| 1 | X | Y |
| 2 | Z | W |

The tuple (∅, P, Q) is **lost** because NULL values do not match in SQL joins.

**Conclusion**

* If **null values exist in the left-hand side of a functional dependency**, **decomposition may no longer be lossless**, as the natural join may **fail to reconstruct** the original relation completely.
* This issue **violates the lossless join property**, leading to **information loss or spurious tuples** when attempting to recover the original relation.
* To ensure a **lossless decomposition**, it's crucial that attributes on the **left-hand side of functional dependencies do not contain null values**.