0.19 Normalization

you-said

Explain How functional dependencies are used to indicate the following: 1 - one to one relationship set exists between entity sets student and instructor 2 - many to one relationship set exists between entity sets student and instructor

ChatGPT said:

Functional dependencies help define relationships between entity sets by showing how attributes are determined by other attributes. Here's how they can be used to express different relationship sets between the entity sets `Student` and `Instructor`:

one-to-one-relationship-between-student-and-instructor

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

In a one-to-one relationship, each student is associated with only one instructor, and each instructor is associated with only one student. This can be expressed using functional dependencies as:

- **Student → Instructor**: Each student determines (is associated with) exactly one instructor.
- **Instructor → Student**: Each instructor determines (is associated with) exactly one student.

This means for every unique student, there is a unique instructor, and vice versa.

many-to-one-relationship-between-student-and-instructor

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

In a many-to-one relationship, multiple students can be associated with the same instructor, but each student is associated with only one instructor. This can be expressed as:

- **Student → Instructor**: Each student determines (is associated with) exactly one instructor.

- **Instructor Student**: This means an instructor does **not** determine a unique student, implying that multiple students can have the same instructor.

Here, the functional dependency indicates that knowing a student uniquely determines their instructor, but knowing an instructor does not uniquely determine a student, since multiple students may share the same instructor.

In summary:

- One-to-One: Both `Student → Instructor` and `Instructor → Student` hold.
- Many-to-One: Only `Student → Instructor` holds, while `Instructor → Student` does not.

you-said-1

Explain this algorithm with example: result := 0: /* 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 at */ for i := 1 to |F| do begin let p → y denote the ith FD; fdcount [1] := IBI: 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 [4] := NIL; for i := 1 to |F| do begin let & → y denote the ith FD; if A € ß then add i to appears [A]; end end addin (a): return (result): procedure addin (a): for each attribute A in a do begin if A € result then begin result := result U {A): for each element i of appears[A] do begin fdcount [i] := fdcount [i] - 1: if fdcount [i] := 0 then begin let B → y denote the ith FD; addin (y): end end end end

ChatGPT said:

This algorithm is used to determine the **closure** of a set of attributes with respect to a given set of functional dependencies (FDs). The closure of a set of attributes is the set of all attributes that can be functionally determined by that set, based on the FDs provided.

Let's break down the algorithm and then illustrate it with an example.

steps-of-the-algorithm

Steps of the Algorithm:

1. **Initialize Variables:**

   - `result := 0` initializes the set of known attributes (those we are certain are in the closure).
   - `fdcount` is an array where each element counts how many attributes on the left side of each FD are not yet known to be in the closure.

- `appears` is an array that keeps track of which FDs an attribute appears on the left-hand side.

2. **Counting and Mapping FDs to Attributes:**

   - For each FD `p → y`, the algorithm calculates how many attributes on the left-hand side (`p`) are not yet known.
   - Then, for each attribute `A`, the algorithm records which FDs have `A` on the left-hand side.

3. **Procedure `addin(a)`:**

   - The procedure adds attributes in `a` to the result set if they are not already in it.
   - It also updates the `fdcount` for each FD that has the newly added attributes on the left-hand side.
   - When all the attributes on the left-hand side of an FD are in the result (i.e., `fdcount[i] = 0`), the right-hand side of that FD is also added to the result (closure).

4. **Result:**

   - After running the algorithm, `result` will contain the closure of the initial set of attributes `a`.

example

Example:

Consider the following set of functional dependencies (FDs) over attributes A, B, C, D, EA,B,C,D,E:

1. $A \to B$A→B
2. $B \to C$B→C
3. $A, C \to D$A,C→D
4. $D \to E$D→E

Let's say we want to compute the closure of $\{A\}${A}.
step-by-step-execution

1. **Initialization:**

   - `result := 0` (initially, the closure is empty).
   - `fdcount[i]` is initialized as follows:

     - `fdcount[1] := 1` (1 attribute on the left of $A \to B$A→B)
     - `fdcount[2] := 1` (1 attribute on the left of $B \to C$B→C)
     - `fdcount[3] := 2` (2 attributes on the left of $A, C \to D$A,C→D)
     - `fdcount[4] := 1` (1 attribute on the left of $D \to E$D→E)

2. **Build `appears`:**

   - `appears[A] := {1, 3}` (since AA appears on the left-hand side of FD 1 and FD 3).
   - `appears[B] := {2}` (since BB appears on the left-hand side of FD 2).
   - `appears[C] := {3}` (since CC appears on the left-hand side of FD 3).
   - `appears[D] := {4}` (since DD appears on the left-hand side of FD 4).

3. **Invoke `addin(A)`:**

   - Add AA to `result`. Now, `result = {A}`.
   - For each FD where AA appears (FD 1 and FD 3):

     - Decrement `fdcount[1]` to 0. Since `fdcount[1] = 0`, invoke `addin(B)` (because $A \to B$A→B).
     - Decrement `fdcount[3]` to 1 (since $A, C \to D$A,C→D still requires CC).

4. **Invoke `addin(B)`:**

   - Add BB to `result`. Now, `result = {A, B}`.
   - For FD 2 (because $B \to C$B→C):

     - Decrement `fdcount[2]` to 0. Since `fdcount[2] = 0`, invoke `addin(C)`.

5. **Invoke `addin(C)`:**

    - Add CC to `result`. Now, `result = {A, B, C}`.
    - For FD 3 (because $A, C \to D$A,C→D):

        - Decrement `fdcount[3]` to 0. Since `fdcount[3] = 0`, invoke `addin(D)`.

6. **Invoke `addin(D)`:**

    - Add DD to `result`. Now, `result = {A, B, C, D}`.
    - For FD 4 (because $D \to E$D→E):

        - Decrement `fdcount[4]` to 0. Since `fdcount[4] = 0`, invoke `addin(E)`.

7. **Invoke `addin(E)`:**

    - Add EE to `result`. Now, `result = {A, B, C, D, E}`.

final-result

Final Result:

The closure of $\{A\}${A} is $\{A, B, C, D, E\}${A,B,C,D,E}.
summary-of-the-algorithms-flow

Summary of the Algorithm's Flow:

- The algorithm systematically processes each FD, adding attributes to the closure when the left-hand side of the FD is fully included in the closure.
- It continues recursively, using the `addin` procedure to track dependencies and ensure all transitive closures are accounted for.

you-said-2

How does this query tests that a functional dependency B -> C exists for Database Schema R(A, B, C) on relation r: Select B From r Group By B Having Count(Distinct C) > 1;

ChatGPT said:

The query provided is used to **test whether a functional dependency** $B \rightarrow C$**B→C holds** in a relation `r` with schema $R(A, B, C)$R(A,B,C). A functional dependency $B \rightarrow C$B→C means that if two rows in the relation `r` have the same value for attribute `B`, they must also have the same value for attribute `C`.

Let's break down the query and its logic:

query

Query:

[] sqlCopy codeSelect B From r Group By B Having Count(Distinct C) > 1;

explanation-of-the-query

Explanation of the Query:

1. `Select B`:
   The query retrieves the value of attribute `B`.
2. `From r`:
   The query operates on relation `r`, which has the schema $R(A, B, C)$R(A,B,C).
3. `Group By B`:
   This clause groups the rows of the relation by the value of attribute `B`. Essentially, all rows with the same value for `B` are grouped together.
4. `Having Count(Distinct C) > 1`:
   After grouping by `B`, the `Having` clause filters the groups. It checks if there is more than one distinct value of `C` in the group for a particular value of `B`.

   - `Count(Distinct C)` counts the number of distinct values of `C` for each group of rows with the same value of `B`.
   - `Having Count(Distinct C) > 1` ensures that only groups where `B` is associated with more than one distinct value of `C` are selected.

logic-behind-testing-the-functional-dependency

Logic Behind Testing the Functional Dependency:

- **If** $B \to C$**B→C holds**, then for every value of `B`, there must be **exactly one corresponding value of** `C`. This means that for every group of rows with the same `B` value, there should be only one distinct value of `C`.
- **If** $B \to C$**B→C does not hold**, there will be some value of `B` for which there are **multiple distinct values of** `C`. This violates the functional dependency $B \to C$B→C.

what-the-query-returns

What the Query Returns:

- The query returns the values of `B` for which there is more than one distinct value of `C`.
- **If the query returns any rows**, this means that $B \to C$B→C **does not hold** because there is at least one value of `B` that maps to multiple values of `C`.
- **If the query returns no rows**, this means that for every value of `B`, there is only one corresponding value of `C`, which indicates that $B \to C$B→C **holds**.

example-1

Example:

Consider a relation `r` with the following data:

A	B	C
1	1	2
2	1	2
3	2	3
4	2	4

- If you group by `B`, you get two groups:
  - For `B = 1`, the distinct values of `C` are {2} (only one distinct value).

  - For `B = 2`, the distinct values of `C` are {`3, 4`} (two distinct values).
- The query will return `B = 2` because it has more than one distinct value of `C`. Therefore, in this case, the functional dependency $B \to C$B→C **does not hold**.

summary

Summary:

- The query tests whether the functional dependency $B \to C$B→C holds.
- **If the query returns no rows**, $B \to C$B→C holds (each `B` determines exactly one `C`).
- **If the query returns one or more rows**, $B \to C$B→C does not hold (some `B` values determine more than one `C` value).

you-said-3

How to create an assertion for this check?
ChatGPT said:
To create an **assertion** that ensures the functional dependency $B \to C$B→C holds for the relation `r` in a database schema $R(A, B, C)$R(A,B,C), we can use SQL's `ASSERTION` mechanism. However, many databases like MySQL, PostgreSQL, and others do not natively support `CREATE ASSERTION`. Instead, you can simulate this behavior using **triggers** or **constraints**.

Since SQL assertions are not widely supported, I will first explain the conceptual assertion and then provide a practical alternative using a **trigger**.
conceptual-assertion-if-supported-by-the-dbms

Conceptual Assertion (if supported by the DBMS):

If your database supports `CREATE ASSERTION`, you can express the check as follows:
[] sqlCopy codeCREATE ASSERTION fd_check CHECK (NOT EXISTS (SELECT B FROM r GROUP BY B HAVING COUNT(DISTINCT C) > 1));
explanation
Explanation:

- This assertion ensures that there does not exist any `B` value that maps to more than one distinct `C` value.
- The `NOT EXISTS` clause checks that no group of `B` has more than one distinct `C`.

practical-alternative-using-triggers-for-databases-without-create-assertion

Practical Alternative: Using Triggers (for databases without `CREATE ASSERTION`)Practical Alternative: Using Triggers (for databases without CREATE ASSERTION)

Since many databases do not support `CREATE ASSERTION`, you can implement the check using **triggers** to enforce the constraint whenever data is inserted or updated.

Here's how you could create a trigger in PostgreSQL to enforce that $B \rightarrow C$B→C holds:

step-1-create-a-trigger-function

[] sqlCopy codeCREATE OR REPLACE FUNCTION check_fd_b_to_c() RETURNS TRIGGER AS

$BEGIN--CheckifthereexistsanyBthathasmorethanonedistinctCIFEXISTS(SELECTBFROMrGRO$

RETURN NEW; END;

$$LANGUAGEplpgsql;$$

step-2-create-a-trigger

[] sqlCopy codeCREATE TRIGGER fd_b_to_c_trigger AFTER INSERT OR UPDATE ON r FOR EACH STATEMENT EXECUTE FUNCTION check_fd_b_to_c();

how-it-works

How It Works:

- **Trigger Function** (`check_fd_b_to_c`): This function checks if any value of `B` has more than one distinct `C`. If it finds any violation, it raises an exception to prevent the insert or update.
- **Trigger** (`fd_b_to_c_trigger`): This trigger executes the function after every `INSERT` or `UPDATE` operation on the relation `r`. If the functional dependency is violated, the operation is aborted.

example-2

Example:

If you try to insert the following rows into relation `r`:

[] sqlCopy codeINSERT INTO r (A, B, C) VALUES (1, 1, 2); -- Success INSERT INTO r (A, B, C) VALUES (2, 1, 3); -- Error: Functional dependency B -> C is violated

The second `INSERT` statement would fail because it violates the functional dependency $B \rightarrow C$B→C, as `B = 1` has two different `C` values (`2` and `3`).

summary-1

Summary:

- While `CREATE ASSERTION` is conceptually the most direct way to enforce functional dependencies, it is not widely supported.
- A **trigger-based** solution is a practical alternative that ensures functional dependencies are respected during `INSERT` and `UPDATE` operations by enforcing the check dynamically.

prompt-textarea
composer-background