Functional Dependencies

Functional Dependencies

- A functional dependence is a **constraint between two sets of attributes** from the database.
- It plays major role in **differentiating** good database design from bad database design.
- Functional dependency is a type of constraint that is generalization of notion of key.
- Bad database design
  - Repetition of information
  - Inability to represent certain information
- Good database
  - Avoid redundant data
  - Ability to represent all information and relationship among attributes.

Definition

- A function dependency denoted by **X → Y**, between two sets of attributes X and Y that are subset of R specifies a constraints on the possible tuples that can form a relation state r of R.
- The value of component **Y depends on values of component X**
- Abbreviation for functional dependencies is **FD or f.d.**
- The set of attributes X is called the left-hand side of the FD, and Y is called the right-hand side.

Example

- Suppose we have a student table with attributes: Stu_Id, Stu_Name, Stu_Age.
- Here **Stu_Id attribute uniquely identifies the Stu_Name** attribute of student table because if we know the student id we can tell the student name associated with it.
- This is known as functional dependency.
- Can be written as **Stu_Id->Stu_Name** or in words we can say Stu_Name is functionally dependent on Stu_Id.

Types of Functional Dependency

- Functional Dependency has three forms:
  - Trivial Functional Dependency
  - Non-Trivial Functional Dependency

Trivial Functional Dependency

It occurs when B is a subset of A in:

A ->B

Example
We are considering the same **<Department>** table with two attributes to understand the concept of trivial dependency.

The following is a trivial functional dependency since **DeptId** is a subset of **DeptId** and **DeptName**

{ DeptId, DeptName } -> Dept Id

Non –Trivial Functional Dependency
It occurs when B is **not a subset** of A in:

A ->B

Example

DeptId -> DeptName

The above is a non-trivial functional dependency since DeptName is a not a subset of DeptId.

An employee table with three attributes: emp_id, emp_name, emp_address.
The following functional dependencies are non-trivial:

emp_id -> emp_name (emp_name is not a subset of emp_id)
emp_id -> emp_address (emp_address is not a subset of emp_id)

Non-loss Decomposition
What is decomposition?

- Decomposition is the process of **breaking down in parts or elements**.
- It replaces a relation with a collection of smaller relations.
- It breaks the **table into multiple tables in a database**.
- It should always be lossless, because it confirms that the information in the original relation can be accurately reconstructed based on the decomposed relations.
- If there is no proper decomposition of the relation, then it may lead to problems like loss of information.

Properties of Decomposition

Following are the properties of Decomposition,
1. Lossless Decomposition
2. Dependency Preservation
3. Lack of Data Redundancy

1. Lossless Decomposition

- Decomposition must be lossless. It means that the **information should not get lost from the relation** that is decomposed.
- It gives a guarantee that the join will result in the same relation as it was decomposed.
    - Let R be a relation schema and let F be a set of functional dependencies on R.
    - Let 'R1' & 'R2' form a decomposition of R.
    - Let r(R) be a relation with schema R.
    - Decomposition is a losses decomposition, if for legal database instance
        - $\prod_{R1}(r) \bowtie \prod_{R2}(r) = r$
    - If user project r onto R1 & R2, and compute the natural join of the projection results exactly 'r'. Hence no loss, non-loss decomposition.

Example:

Let's take 'E' is the Relational Schema, With instance 'e'; is decomposed into: E1, E2, E3, ... En; With instance: e1, e2, e3, ... en, If e1 ⋈ e2 ⋈ e3 . . . . ⋈ en, then it is called as **'Lossless Join Decomposition'.**

- In the above example, it means that, if **natural joins of all the decomposition give the original relation, then it is said to be lossless join decomposition**.

Example: <Employee_Department> Table

Eid	Ename	Age	City	Salary	Deptid	DeptName
E001	ABC	29	Pune	20000	D001	Finance
E002	PQR	30	Pune	30000	D002	Production
E003	LMN	25	Mumbai	5000	D003	Sales
E004	XYZ	24	Mumbai	4000	D004	Marketing
E005	STU	32	Bangalore	25000	D005	Human Resource

- Decompose the above relation into two relations to check whether a decomposition is lossless or lossy.
- Now, decompose the relation that is Employee and Department.

Relation 1 : <Employee> Table

Eid	Ename	Age	City	Salary
E001	ABC	29	Pune	20000
E002	PQR	30	Pune	30000
E003	LMN	25	Mumbai	5000
E004	XYZ	24	Mumbai	4000
E005	STU	32	Bangalore	25000

Employee Schema contains (Eid, Ename, Age, City, Salary).

Relation 2 : <Department> Table

Deptid	Eid	DeptName

D001	E001	Finance
D002	E002	Production
D003	E003	Sales
D004	E004	Marketing
D005	E005	Human Resource

Department Schema contains (Deptid, Eid, DeptName).

- So, the above decomposition is a Lossless Join Decomposition, because the two relations contains one common field that is 'Eid' and therefore join is possible.
- Now apply natural join on the decomposed relations.

Employee ⋈ Department

Eid	Ename	Age	City	Salary	Deptid	DeptName
E001	ABC	29	Pune	20000	D001	Finance
E002	PQR	30	Pune	30000	D002	Production
E003	LMN	25	Mumbai	5000	D003	Sales
E004	XYZ	24	Mumbai	4000	D004	Marketing
E005	STU	32	Bangalore	25000	D005	Human Resource

Hence, the decomposition is Lossless Join Decomposition.

- If the <Employee> table contains (Eid, Ename, Age, City, Salary) and <Department> table contains (Deptid and DeptName), then it is not possible to join the two tables or relations, because there is no common column between them. And it becomes **Lossy Join Decomposition.**

2. Dependency Preservation

- Dependency is an important constraint on the database.
- Every dependency **must be satisfied by at least one decomposed table**.
- If {A → B} holds, then two sets are functional dependent. And, it becomes more useful for checking the dependency easily if both sets in a same relation.
- This decomposition property can only be **done by maintaining the functional dependency**.
- In this property, it allows to check the updates without computing the natural join of the database structure.
- Set of restriction F1, F2, F3… Fn is the set of dependencies that can be checked efficiently.
    - Let, **F′ = F1 U F2 U F3 U ….U Fn**.
    - F′ is a set of functional dependencies on schema R but in general F′ ≠ F.
    - The property **$F'^{+} = F^{+}$** is a dependency preserving decomposition.
    - F = **{A → B, B → C}. F^{+} is dependency A → C even though it is not in F.**

1. Lack of Data Redundancy

- Lack of Data Redundancy is also known as a **Repetition of Information.**
- The proper decomposition should not suffer from any data redundancy.
- The careless decomposition may cause a problem with the data.
- The lack of data redundancy property may be achieved by Normalization process.

Normalization

- **Normalization** is a process of organizing the data in database to avoid
  - Data Redundancy
  - Insertion anomaly
  - Deletion anomaly.
  - Update anomaly &
- Normalization divides the **larger table into the smaller table** and links them using relationship.
- The normal form is used to reduce redundancy from the database table.
- **Normalization** is the process of **minimizing redundancy** from a relation or set of relations. Redundancy in relation may cause insertion, deletion and updation anomalies.
- Normalization is a systematic approach of decomposing tables to eliminate data redundancy (repetition) and undesirable characteristics like Insertion, Update and Deletion Anamolies.

Anomalies

- **Insert anomalies –** When user tried to insert data in a record that does not exist at all.
- **Deletion anomalies –** When user tried to delete a record, but parts of it was left undeleted because of unawareness, the data is also saved somewhere else. When user tried to delete a record, which cause deletion of some other data from the table/realation.
- **Update anomalies** − If data items are scattered and are not linked to each other properly, then it could lead to strange situations. For example, when we try to update one data item having its copies scattered over several places, a few instances get updated properly while a few others are left with old values. Such instances leave the database in an inconsistent state.
- **Example**: Suppose a manufacturing company stores the employee details in a table named employee that has four attributes:
  - emp_id for storing employee's id,
  - emp_name for storing employee's name,
  - emp_address for storing employee's address and
  - emp_dept for storing the department details in which the employee works.

emp_id	emp_name	emp_address	emp_dept
101	Rick	Delhi	D001
101	Rick	Delhi	D002
123	Maggie	Agra	D890
166	Glenn	Chennai	D900
166	Glenn	Chennai	D004

- The above table is not normalized. We will see the problems that we face when a table is not normalized.
- **Insert anomaly**: Suppose a new employee joins the company, who is under training and currently not assigned to any department then we would not be able to insert the data into the table if emp_dept field doesn't allow nulls.
- **Delete anomaly**: Suppose, if at a point of time the company closes the department D890 then deleting the rows that are having emp_dept as D890 would also delete the information of employee Maggie since she is assigned only to this department.
- **Update anomaly**: In the above table we have two rows for employee Rick as he belongs to two departments of the company. If we want to update the address of Rick then we have to update the same in two rows or the data will become inconsistent. If somehow, the correct address gets updated in one department but not in other then as per the database, Rick would be having two different addresses, which is not correct and would lead to inconsistent data.
- To overcome these anomalies we need to normalize the data