Electronics Engineering

Computer Organization and Architecture

Comprehensive Theory

with Solved Examples and Practice Questions

MADE EASY Publications

MADE EASY Publications

Corporate Office: 44-A/4, Kalu Sarai (Near Hauz Khas Metro Station), New Delhi-110016
E-mail: infomep@madeeasy.in
Contact: 011-45124660, 8860378007

Visit us at: www.madeeasypublications.org

Computer Organization and Architecture

First Edition: 2015
Second Edition: 2016
Third Edition: 2017
Fourth Edition: 2018
Fifth Edition: 2019
Sixth Edition: 2020

Contents

Computer Organization and Architecture

■■■

CHAPTER 4

Database Management System

4.1 Introduction

A **database** is a collection of related data. By **data**, we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, e-mail address, and office addresses of the people in office.

- A database represents some aspect of the real world, sometimes called the **miniworld** or the **universe of discourse (UoD)**.
- A database is a logically coherent collection of data with some inherent meaning.
- A database is designed, built, and populated with data for a specific purpose.

A **database management system (DBMS)** is a collection of programs that enables users to create and maintain a database. The DBMS is a *general-purpose software sys-tem* that facilitates the processes of *defining, constructing, manipulating,* and *sharing* databases among various users and applications. A database can be defined by involving specification or the data types, structures, and constraints of the data to be stored in the data-base. The database definition or descriptive information is stored by the DBMS in the form of a database catalog or dictionary; which is known as **meta-data**. **Constructing** the database is the process of storing the data on some storage medium that is con-trolled by the DBMS. **Manipulating** a database includes functions such as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data. **Sharing** a database allows multiple users and programs to access the database simultaneously.

An **application program** accesses the database by sending queries or requests for data to the DBMS. A **query** typically causes some data to be retrieved; a **transaction** may cause some data to be read and some data to be written into the database.

4.2 Introduction to Database Design

Entity relationships (*ER*) model is high level database design allows us to describe the data involved in real-world enterprise in terms of objects and their relationships and is widely used to develop initial database design. In overall design process, the *ER* model is used in a phase called conceptual database design.

4.2.1 Database Design and ER Diagrams

The database design process can be divided into six steps. The *ER* model is most relevant to the first three steps.

Requirements Analysis

The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. In other words, we must find out what the users want from the database. This is usually an informal process that involves discussions with user groups, a study of the current operating environment and how it is expected to change, analysis of any available documentation on existing applications that are expected to be replaced or complemented by the database, and so on.

Conceptual Database Design

The information gathered in the requirements analysis step is used to develop a high-level description of the data to be stored in the database, along with the constraints known to hold over this data. This step is often carried out using the *ER* model and is discussed in the rest of this chapter. The *ER* model is one of several high-level, or semantic, data models used in database design. The goal is to create a simple description of the data that closely matches how users and developers think of the data

Logical Database Design

We must choose a DBMS to implement our database design, and convert the conceptual database design into a database schema in the data model of the chosen DBMS. We will consider only relational DBMSs, and therefore, the task in the logical design step is to convert an *ER* schema into a relational database schema.

4.2.2 Entity, Attributes, Entity Set

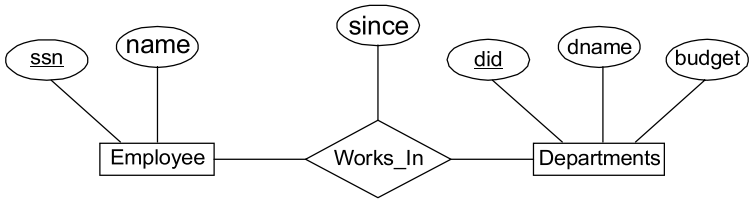
- Entity is an object that exist and is distinguishable from other objects. For example a person with give UID is an entity as he can be uniquely identified as one particular person.
- An entity may be concrete (person) or abstract (job)
- An entity set is a collection of similar entities. (All persons having an account at a bank)
- Entity sets need not be disjoint. For example, the entity set employee (all employees of a bank) and entity set customer (all customers of the bank) may have members (Entity) in common.
- An entity is described using a set of attributes, all entities in a given entity set have same attributes, this is what we mean by similar.
- For each attribute associated with an entity set, we must identify domain of attribute which is the set of permitted values (*e.g.* if a company rates employees on a scale of 1 to 10 and stores rating in a field called rating, the associated domain consist of integers 1 through 10)
- An analogy can be made with the programming language notion of type definition, concept of entity set corresponds to the programming language type definition.
- A variable of a given type has a particular value at a time, thus a programming language variable corresponds to an entity in ER model.

4.2.3 Relationship and Relationship Sets

- Relationship is association between two or more entities
- Relationship set is a set of relationships of same type *i.e.* relate two or more entity sets

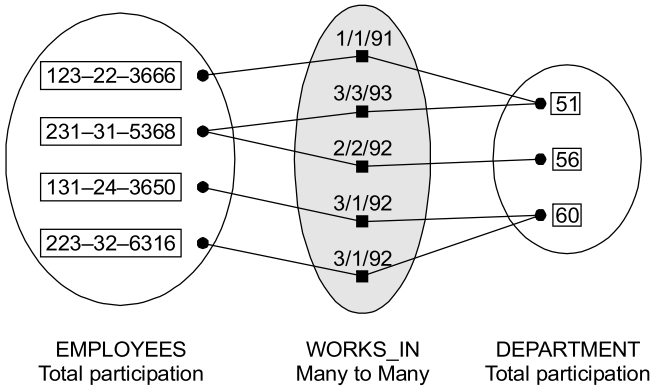
A relationship set can be thought of as a set of n-tuples: $\{(e_1,\ldots,e_n) \mid e_1 \in E_1,\ldots, e_n \in E_n\}$

Each n-tuple denotes a relationship involving n entities e_1 through e_n, where entity e_i is in entity set E_i. In Figure. We show the relationship set Works_In, in which each relationship indicates a department in which an employee works. Note that several relationship sets might involve the same entity sets. For example, we could also have a Manages relationship set involving Employees and Departments.



A relationship can also have **descriptive attributes.** Descriptive attributes are used to record information about the relationship, rather than about any one of the participating entities.

An **instance** of a relationship set is a set of relationships. Intuitively, an instance can be thought of as a 'snapshot' of the relationship set at some instant in time. An instance of the Works_In relationship set is shown in Figure. Each Employees entity is denoted by its *ssn,* and each Departments entity is denoted by its did, for simplicity. The *since* value is shown beside each relationship.



An Instance of the Works_In Relationship Set

4.2.4 Relationship Constraints

There are two types of relationship constraints

- Participation constraints
- Cardinality ratio.

There are two types of participation constraints:

(*i*) **Total participation constraints (existence dependency):** The participation of an entity set *E* in a relationship set *R* is said to be total if every entity in *E* participates in at least one relationship in *R*. This participation is displayed as a double line connecting.

Example: If a company policy states that every employee must work for a department, then an employee entity can exist only if it participates in at least one "worksfor" relationship instance.

(*ii*) **Partial Dependency:** If only some entities in *E* participate in relationship in *R*, the participation of entity set *E* in relationship *R* is said to be partial. This participation is displayed as a single line connecting.

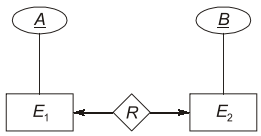
Example: Not every employee "Manages" a department.

Mapping Constraints: The possible cardinality ratio for binary relationship are:

- One to one (1 : 1)
- One to many (1 : *M*)
- Many to one (*M* : 1)
- Many to many (*M* : *M*)

One to One (1 : 1): An entity (tuple) in E_1 is associated with atmost one Entity (tuple) in E_2, and an entity in E_2 is associated with atmost one entity in *A*

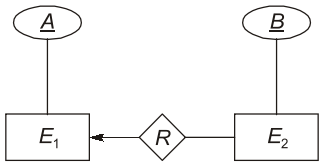
Example:



Candidate keys of relation *R* (*A*, *B*) = *A*, *B*

One to Many (1 : *M*): An entity (tuple) in E_1 is associated with zero or more entities in E_2 but an entity in E_2 can be associated with at most one entity in E_1

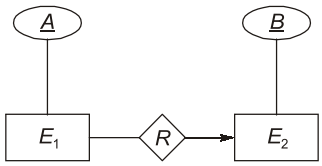
Example :



Candidate key of *R* (*A*, *B*) = *B*

Many to one (*M* : 1): An entity (tuple) in E_2 is associated with zero or more entities in E_1 but an entity in E_1 can be associated with at most one entity in E_2.

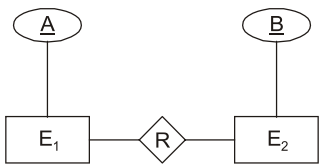
Example :



Candidate key of *R* (*A*, *B*) = *A*

Many to many (*M* : *M*): An entity (tuple) in E_2 is associated with zero or more entities in E_1 and An entity (tuple) in E_1 is associated with zero or more entities in E_2.

Example :

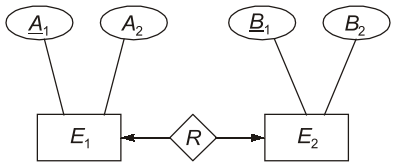


Candidate key of *R* (*A*, *B*) = *AB*

4.2.5 Minimization of ER Diagram

1. **1 : 1 cardinality** with partial participation at both end

   Example:



   Consider the relation instances of relation E_1, E_2 and R.

E_1:

A_1	A_2
1	P
2	P
3	q

E_2:

B_1	B_2
11	P
21	q
31	R

R:

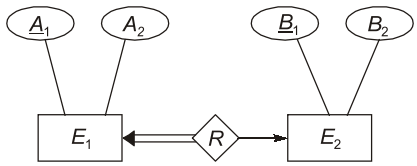
A_1	B_1
1	11
3	21

   E_1 and R can be combined to form a single table with A_1 is primary key (Unique and not NULL) and B_1 as alternate key as well as foreign key. Similarly R can be combined with E_2. But can't be combined to both E_1 and E_2 *i.e.* a single table E_1RE_2

   - If R is combined with both E_1 and E_2 i.e. a single relation E_1RE_2 then it has no primary key
   - **Minimum number of tables** = 2

     (E_1R) and E_2 or E_1 and (E_2R)

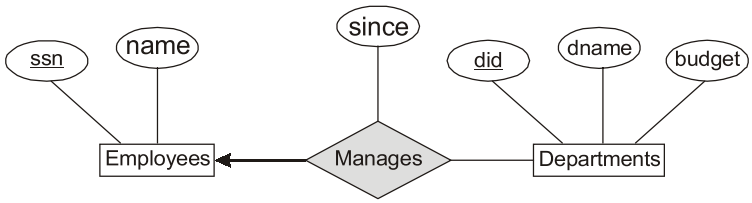
2. **1 : 1 cardinality** with total participation atleast one side. The relationship set E_1RE_2 (A_1, A_2, $\underline{B_1}$, B_2) has $B1$ as primary key and A_1 as alternate key.



   Note that A_1 can be null because there may be an entity in E_2 which is not related to any entity of E_1.

   - **Minimum number of tables** = 1 *i.e.* E_1RE_2

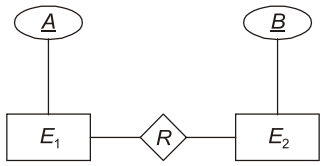
3. **1 : *M* Cardinality:** Consider relationship set called manages between the Employees and Departments entity sets such that each department has at most one manager, although a single employee is allowed to manage more than one department. The restriction that each department entity appears in at most one manager is an example of a key constraint, and it implies that each departments entity appears in at most one manages relationship in any allowable instance of Manages. This restriction is indicated in the *ER* diagram of Figure.



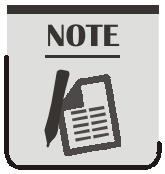
The entity set Departments and relationship set Manages combined into a single entity set dept_manages (did, dname, Eid, Since) with did as primary key and Eid, is foreign key to the entity set Employees.

- If foreign key attribute Eid is NULL, then partial participation from Employees set.
- Minimum number of tables = 2

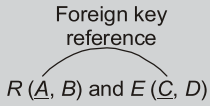
4. ***M*** **:** ***M*** **Cardinality:** Consider the following *ER* model



Relationship *R* has primary key as *AB*. *R* can't be combined with E_1 or E_2.
Minimum number of tables = 3



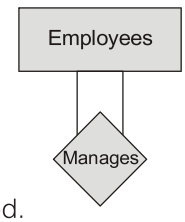
When to minimize?
If relationship set *R* is having a key as *A* which is also foreign key referencing to entity set *E*, then *R* and *E* combined to a single entity set e.g.



R and *E* combined to a single table = *RE* (*C*, *B*, *D*)

4.2.6 Self Referential Relationship

Relationship set relates to same entity sets *i.e.* pair of entities relating to each other.



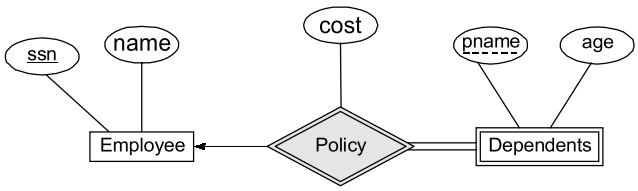
Example: Each employee manages more than one employee but a employee has only one manager *i.e.* 1 : *M* relation exist. Consider employee has attributes Eid, Ename, and Manages have attribute Sup*ID*, Sub*ID*, both Sup*ID* and Sub*ID* are foreign key in the employee referencing Eid, if 1 : *M* relationship exist, Sub*ID* is the key in relationship set manages.

Therefore combined into a single table. If *M* : *M* relationship then two tables are required.

4.2.7 Weak Entity Set

Thus far, we have assumed that the attributes associated with an entity set include a key. This assumption does not always hold. For example, suppose that employees can purchase insurance policies to cover their dependents. We wish to record information about policies, including who is covered by each policy, but this information is really our only interest in the dependents of an employee. If an employee quits, any policy owned by the employee is terminated and we want to delete all the relevant policy and dependent information from the database.

We might choose to identify a dependent by name alone in this situation, since it is reasonable to expect that the dependents of a given employee have different names. Thus the attributes of the Dependents entity set might be *pname* and *age.* The attribute *pname* does *not* identify a dependent uniquely. Recall that the key for Employees is *ssn;* thus we might have two employees called *XYZ* and each might have a son called *ABC*.



Dependents is an example of a **weak entity set.** A weak entity can be identified uniquely only by considering some of its attributes in conjunction with the primary key of another entity, which is called the **identifying owner**. The following restrictions must hold:

1. The owner entity set and the weak entity set must participate in a one-to-many relationship set (one owner entity is associated with one or more weak entities, but each weak entity has a single owner). This relationship set is called the **identifying relationship set** of the weak entity set.
2. The weak entity set must have total participation in the identifying relationship set.

4.3 Introduction to Database Design and Normalization

Database design may be performed using two approaches: **bottom-up or top-down**. A **bottom-up** design methodology considers the basic relationships among individual attributes as the starting point and uses those to construct relation schemas. This approach is not very popular in practice because it suffers from the problem of having to collect a large number of binary relationships among attributes as the starting point. For practical situations, it is next to impossible to capture binary relationships among all such pairs of attributes. On the other hand, a top-down design methodology starts with a number of groupings of attributes into relations that exist together naturally. The relations are then analyzed individually and collectively, leading to further decomposition until all desirable properties are met.

Primary Key

Key (primary key) of a relation schema is the minimal set of attributes that uniquely identifies each tuple (row) in the relation which has non-NULL values.

If a relation schema has more than one key, each is called candidate key. One of the candidate key is designated to be primary key and others are called **secondary keys (Alternate keys)**. Alternate keys allowed NULL values.

Super Key

A superkey of a relation schema $R = \{A_1, A_2, \ldots, A_n\}$ is set of attributes $S \subseteq R$ with the property that no two tuples, t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$. A key K is a superkey with the additional property that removal of any attribute from K will cause K not to be superkey any more.

The difference between a key and a superkey is that a key has to be minimal; that is, if we have a key $K = \{A_1, A_2, \ldots, A_k\}$ of R, then $K - \{A_i\}$ is not a key of R for any A_i, $1 \le i \le k$.

***Example* - 4.1** Relation $R(A_1, A_2, \ldots A_n)$ with A_1 as the primary key. Then how many super keys possible?

Solution:

A candidate key remaining $A_2\ A_3 \ldots A_n$ any subset of attribute which combine with A_1 is superkey.

Total keys = 2^{n-1}.

Foreign Key

Foreign key is the set of attribute references to the primary key or alternate key of same table or some other table.

4.3.1 Integrity Constraints

An Integrity Constrains (IC) is a condition specified on a database schema and restricts the data that can be stored in an instance of the database. If a database instance satisfies all the integrity constrains specified on the database schema it is legal instance. A DBMS enforces IC, in that it permits only legal instances to be stored in the database.

Integrity constrains are specified and enforces at different times.

(*i*) When a DBA or end user defines a database schema, he or she specifies the *IC*s that must hold on any instance of this database.

(*ii*) When a database application is run, the DBMS checks for violations and disallows changes *IC*s. It is important to specify when *IC*s are checked, i.e., when a data is inserted, deleted or updated in the table.

Examples of integrity constraints are: Domain constraints, Referential Integrity Constraints, Function dependencies (FDs), Assertions and Triggers.

4.3.2 Domain Constraints

A relation schema specifies the domain of each field or column in the relation. The values that appear in a column must be drawn from the domain associated with that column. The domain of a field is essentially the type of that field, in programming language terms, and restricts the values that can appear in the field. The check clause in SQL permits the schema designer to specify a predicate (condition) that must be satisfied by value assigned to a variable whose type is the domain.

4.3.3 Referential Integrity Constraints (RIC)

Sometimes the information stored in a relation is linked to the information stored in another relation. If one of the relations is modified, the other must be checked and perhaps modified to keep the data consistent. An IC involving both relations must be specified if a DBMS is to make such checks. The most common *IC* involving two relations is a **foreign key constraint** or **RIC**.

Suppose we have two relations

Student (Sid, name, gpa)

Enrolled (studid, cid, grade)

Where studid in Enrolled references to the primary key Sid in the students relation. To ensure that only bonafide students can enroll in courses, any value that appears in the studid field of an instance of the Enrolled relation should also appear in the sid field of some tuple in the Students relation. The studid field of Enrolled is called a **foreign key** and **refers** to Students. The foreign key in the referencing relation (Enrolled, in our example) must match the primary key of the referenced relation (Students); that is, it must have the same number of columns and compatible data types, although the column names can be different.

Every value of the referencing attribute (Studid) must be null or available in the referenced attribute (sid) *i.e.*, Studid is subset of Sid. Finally note that a foreign key could refer to the same relation.

Enforcing RIC

SQL provides several alternative ways to handle foreign key violations. We must consider three basic questions:

1. What should we do if an Enrolled row is inserted, with a studid column value that does not appear in any row of the Students table? In this case, the Insert command is simply rejected.
2. What should we do if a Students row is deleted?
   The options are:
   - Delete all Enrolled rows that refer to the deleted students row.
   - Disallow the deletion of the students row if an enrolled row refers to it.
   - Set the studid column to the sid of some (existing) 'default' student, for every enrolled row that refers to the deleted students row.
   - For every enrolled row that refers to it, set the studid column to null. In our example, this option conflicts with the fact that studid is part of the primary key of enrolled and therefore cannot be set to null. Therefore, we are limited to the first three options in our example, although this fourth option (setting the foreign key to null) is available in general.

3. What should we do if the primary key value of a students row is updated?
   The options here are similar to the previous case.
   SQL allows us to choose any of the four options on DELETE and UPDATE. For example, we can specify that when a students row is deleted, all enrolled rows that refer to it are to be deleted as well, but that when the sid column of a students row is modified, this update is to be rejected if an enrolled row refers to the modified students row:

```
CREATE TABLE Enrolled    ( studid CHAR (20),
                           cid  CHAR (20),
                           grade CHAR (10),
                          PRIMARY KEY (studid, cid),
                          FOREIGN KEY (studid) REFERENCES Students (Sid)
                                ON DELETE CASCADE
                                ON UPDATE NO ACTION)
```

The options are specified as part of the foreign key declaration. The default option is NO ACTION, which means that the action (DELETE or UPDATE) is to be rejected. Thus, the ON UPDATE clause in our example could be omitted, with the same effect. The CASCADE keyword says that, if a Students row is deleted, all enrolled rows that refer to it are to be deleted as well. If the UPDATE clause specified CASCADE, and the sid column of a students row is updated, this update is also carried out in each enrolled row that refers to the updated students row.

If a Students row is deleted, we can switch the enrollment to a 'default' student by using ON DELETE SET DEFAULT. The default student is specified as part of the definition of the sid field in enrolled; for example, sid CHAR (20) DEFAULT '53666'. Although the specification of a default value is appropriate in some situations (*e.g.*, a default parts supplier if a particular supplier goes out of business), it is really not appropriate to switch enrollments to a default student. The correct solution in this example is to also delete all enrollment tuples for the deleted student (that is, CASCADE) or to reject the update.

SQL also allows the use of null as the default value by specifying ON DELETE SET NULL.

4.3.4 Functional Dependency (FD)

A functional dependency (FD) is a kind of *IC* that generalizes the concept of a key. Let R be a relation schema and let X and Y be nonempty sets of attributes in R. We say that an instance r of R satisfies the FD $X \rightarrow Y$. If the following holds for every pair of tuples t_1 and t_2 in r.

If $t_1.X = t_2.X$, then $t_1.Y = t_2.Y$.

$X \rightarrow Y$ is read as X functionally determines Y or simply X determines Y.

An FD $X \rightarrow Y$ essentially says that if two tuples agree on the values in attributes X, they must also agree on the values in attributes Y.

If a constraints on R states that there cannot be more than one tuple with a given X value in any relation instance $r(R)$, that is X is the key of R, however the definition of an FD does not require that the set X be minimal, the additional minimality condition must be met for X to be a key. If $X \rightarrow Y$ holds, where Y is set of all attribute and there is some subset V of X such that $V \rightarrow Y$ holds then X is a super key.

There are two types of FD.

1. **Trivial FD:** If X and Y are attribute set of R and $X \supseteq Y$ then $X \rightarrow Y$ is trivial FD.
   Example: sid $\rightarrow$ sid, Sid Sname $\rightarrow$ sid, Sid Sname $\rightarrow$ Sname
   Every trivial FD implies in relation.
2. **Non-trivial FD:** If X and Y are attribute sets of R and no common attribute between X and Y i.e., $X \cap Y = \phi$ then $X \rightarrow Y$ is non- trivial FD.
   Example: sid $\rightarrow$ gpa, Sname $\rightarrow$ sid gpa
   There may be relation with no non-trivial FD.

4.3.5 Closure of Set of FDs

Set of all *FD*s that include given *FD*s as well as those that can be inferred from the given. *FD*s is called the closure of FDs. If F is the set of given FDs the F^+ is called closure of F.

The following three rules, called Armstrong's Axioms can be applied repeatedly to infer all FDs implied by a set F of FDs. Let X, Y and Z denotes sets of attributes over a relation schema R.

- **Reflexivity:** If $X \supseteq Y$ then $X \rightarrow Y$
- **Augmentation:** If $X \rightarrow Y$ then $XZ \rightarrow YZ$ for any Z
- **Transitivity:** If $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$

Armstrong's Axioms are sound, by sound we mean that given a set of FDs F specified on relational schema R, any dependency that we can infer from F by using Armstrong's Axioms holds in every relation r of R that also complete by complete we mean set of dependencies F^+, which are called closure of F can be determined from F by using Armstrong's Axioms only.

It is convenient to use some additional rule while finding F^+.

- Decomposition or projection rule: If $X \rightarrow YZ$ then $X \rightarrow Y$ and $X \rightarrow Z$
- Union or additive rule: If $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$

4.3.6 Attribute Closure

The algorithm for computing the attribute closure of a set X of attributes is given below:

closure = X;

repeat until there is no change {

if there is an FD $U \rightarrow V$ in F such that $U \subseteq$ closure,

then set closure = closure $\cup$ V

}

This algorithm can be modified to find keys by starting with set X containing a single attribute and stopping as soon as closure contains all attributes in the relation schema. By varying the starting attribute and the order in which the algorithm considers FDs, we can obtain all candidate keys.

***Example* - 4.2** Given a relation $R(A, B, C, D, E, F)$ with FDs

$AB \rightarrow C$

$B \rightarrow D$

$AD \rightarrow E$

Compute $(AB)^+$

Solution:

$(AB)^+ \rightarrow AB$

$A \rightarrow ABC$ $\{AB \rightarrow C\}$

$\rightarrow ABCD$ $\{B \rightarrow D\}$

$\rightarrow ABCDE$ $\{AD \rightarrow E\}$

$(AB)^+ \rightarrow ABCDE$

4.3.7 Membership Test

If we just want to check whether a given dependency $X \rightarrow Y$ is in the closure of set F of FDs. We can do so efficiently without computing F^+ by using closure of X (finding X^+). If X^+ contains Y then $X \rightarrow Y$ is a member of functional dependency set F i.e. $X \rightarrow Y$ is logically implies in F or $F \Rightarrow X \rightarrow Y$.

Example - 4.3 Prove or disprove the following inference rule for functional dependency using (*i*) Armstrong's Axioms (*ii*) Attribute closure

$$\{X \rightarrow Y, XY \rightarrow Z\} \Rightarrow \{X \rightarrow Z\}$$

Solution:

(*i*) **Armstrong's Axioms:**

$X \rightarrow X$ (trivial) and $X \rightarrow Y$ by union rule $X \rightarrow XY$

$X \rightarrow XY$ and $XY \rightarrow Z$ by transitivity $X \rightarrow Z$

(*ii*) **Attribute closure:** If closure of X determines Z in the given FD set then $X \rightarrow Z$ is logically implies in the given FD. $X^+ \rightarrow XY\underline{Z}$

Since X^+ contains Z, $X \rightarrow Z$ implies in the given FDs.

4.3.8 Equivalence of Sets of Functional Dependencies

Two sets of functional dependencies E and F are equivalent if E covers F ($E \supseteq F$) and F covers E ($F \supseteq E$). Therefore equivalence means that every FD in E can be inferred from F and every FD in F can be inferred from E.

E covers *F*	*F* covers *E*	Result
Yes	Yes	$E \equiv F$
Yes	No	$E \supset F$
No	Yes	$F \supset E$
No	No	*E* and *F* not comparable

Example - 4.4 Given below two sets of FDs for a relation $R(A, B, C, D, E)$. Are they equivalent?

$$F1: \{A \rightarrow B, AB \rightarrow C, D \rightarrow AC, D \rightarrow E\}$$
$$F2: \{A \rightarrow BC, D \rightarrow AE\}$$

Solution:

If $F1$ covers $F2$ then every FD in $F2$ logically implies in $F1$.

FDs in $F2$ $A \rightarrow BC$, $D \rightarrow AE$

Check for $A \rightarrow BC$

$(A)^+ \rightarrow A\underline{BC}$ $\{A \rightarrow B, AB \rightarrow C$ in $F1\}$

Check for $D \rightarrow AE$

$(D)^+ \rightarrow D\,\underline{A}\,C\,\underline{E}$ $\{D \rightarrow AC, D \rightarrow E$ in $F1\}$

Hence $F1$ covers $F2$ (i.e., $F1 \supseteq F2$)

If $F2$ covers $F1$ then every FD in F1 logically implies in F2.

$F1$ has 4 FDs $\{A \rightarrow B, AB \rightarrow C, D \rightarrow AC, D \rightarrow E\}$

Check for $A \rightarrow B$

$(A)^+ \rightarrow A\underline{B}$ $\{A \rightarrow BC$ in $F2\}$

Check for $AB \rightarrow C$

$(AB)^+ \rightarrow ABC$ $\{A \rightarrow BC$ in $F2\}$

Check for $D \rightarrow AC$

$(D)^+ \rightarrow D\underline{A}EBC$ $\{D \rightarrow AE, A \rightarrow BC$ in $F2\}$

Check for $D \rightarrow E$

$(D)^+ \rightarrow DA\underline{E}BC$ $\{D \rightarrow AE$ in $F2\}$

Hence $F2$ covers $F1$ i.e., $F2 \supseteq F1$

So $F1 \equiv F2$