Discipline: BS (IT) - 4th Semester
Subject: Database Systems [NEW Course]
Course Code: IT – 411
Final Term Notes: From Week No. 07 – 16
Prepared by: **ARSHAD IQBAL**, Lecturer (CS/IT), ICS/IT - FMCS, The University of Agriculture, Peshawar

Week No. 07: Entity Relationship Model

Entity Relationship Model (E-R Model):

An **Entity Relationship Model** (E-R Model) is a detailed, **logical representation** of the **data** for an **organization** or for business area. An **E – R Diagram** is the **logical design** of the **databases**.

E-R Model is **expressed** in **terms** of **entities** in the business environment, the **relationship** (or association) among those **entities**, and the **attributes** of both the **entities** and their relationship. **ER Model** is used to show the **Conceptual Schema** of an **Organisation**.

Entity Relationship Diagram (ERD) is a **Data Modeling Technique** used in **Databases** and **Software Engineering** to produce a **Conceptual Data Model** of an information system. So, **ERDs** illustrate the **logical structure** of **databases**.

Constructs/Elements in E-R Data Model:
The major activity is identifying entities, attributes, and their relationships to construct model using the **Entity Relationship Diagram**.

The E-R Data Model supports the following major constructs (concepts)/Elements:

Entity → Table:

- Entity Type
- Entity Instance
- Entity Set

Attribute → Column:

- Attribute Domain

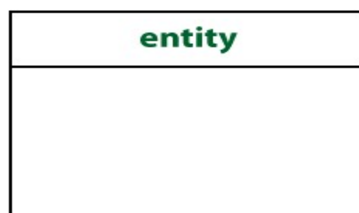
Relationships → line:

- Association

Entity:
An entity is a person, place, thing or event for which data is collected and maintained. For example:

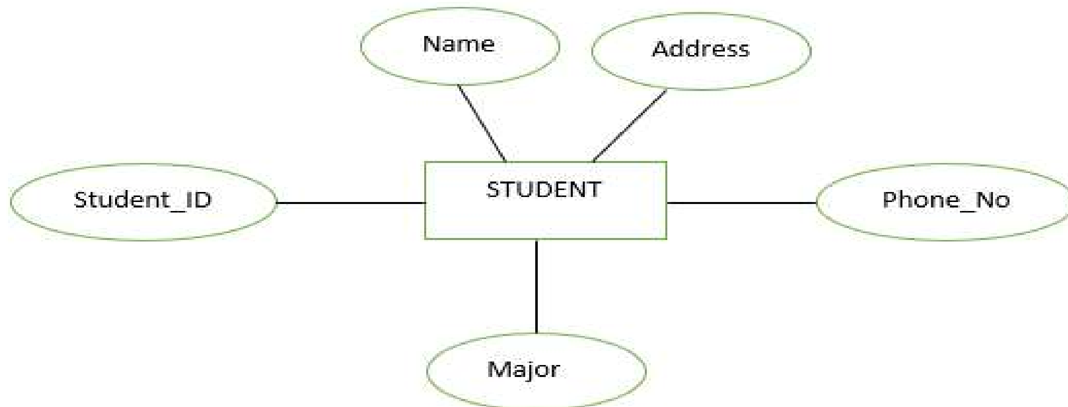
- Person: TEACHER, DOCTOR, PLAYER
- Place: COUNTRY, CITY
- Object: VEHJICLE, TOY, FURNITURE
- Event: PURCHASE, ADMISSION, REGISTRATION
- Concept: ACCOUNT, PROGRAMING

Represented by a rectangle, with its name on the top.



Attributes:

The characteristics of an entity are called **Attributes** or **Properties**. For **example:** A Student Entity Type has attributes like Stuent_id, name, address, phone_number and major etc.



Relationship:

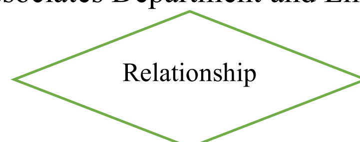
A **Relationship** is a **logical connection** between different **Entity Types**. The **entities** that participate in a **Relationship** are called **participants. Relationship** represents an **association** between two or more **entities**.

An example of a Relationship would be:

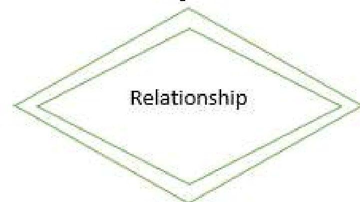
- Employees are assigned to Projects
- Teachers teach the Students
- Projects have Subtasks
- Departments manage one or more Projects

Relationships are the **connections** and **interactions** between the **Entities Instances** e.g., DEPT_EMP associates Department and Employee.

Symbol:



Symbol: For **Relationship** between Strong Entity Type and weak Entity Type.



Another example: A student in a college has many courses and the student is supposed to complete the courses. This creates a many to many Relationships between STUDENT and COURSE.

Entity Relationship Diagrams (ERDs):
E-R Diagram is a graphical representation of E-R model using a set of standard symbols.

Example 01: Draw an ER Diagram for each of the following situations:

1. A company has a number of employees. Each employee may be assigned to one or more projects or may not be assigned to a project. A project must have at least one employee assigned and may have several employees assigned.

2. A hospital patient has a patient history. Each patient has one or more history records. Each patient history record belongs to exactly one patient.

3. An account can be charged against many projects through it may not be charged against any. A project must have at least one accounts charged against it. It may also have many accounts charged against it.

4. An employee must manage exactly one department. A department may or may not have one employee to manage it.

Example 02: Draw an ER Diagram for each of the following situations:

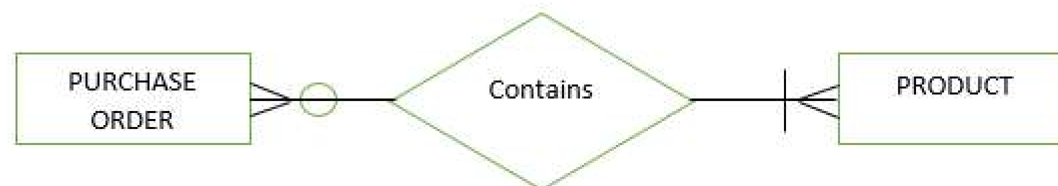
1. A department employs many persons, means a department employee one or many persons. A person is employed by one department at most, means a person may be employed by one department or he may not be employed at all.

2. A manager manages one department at most, means a manager may manage one department or he manages no department. A department is managed by one manager at most, means a department is managed by one manager or it is not managed by any manager.

3. A team consists of many players, means a team employs at least one player or many players. A player plays for one team, means a player plays for exactly one team.

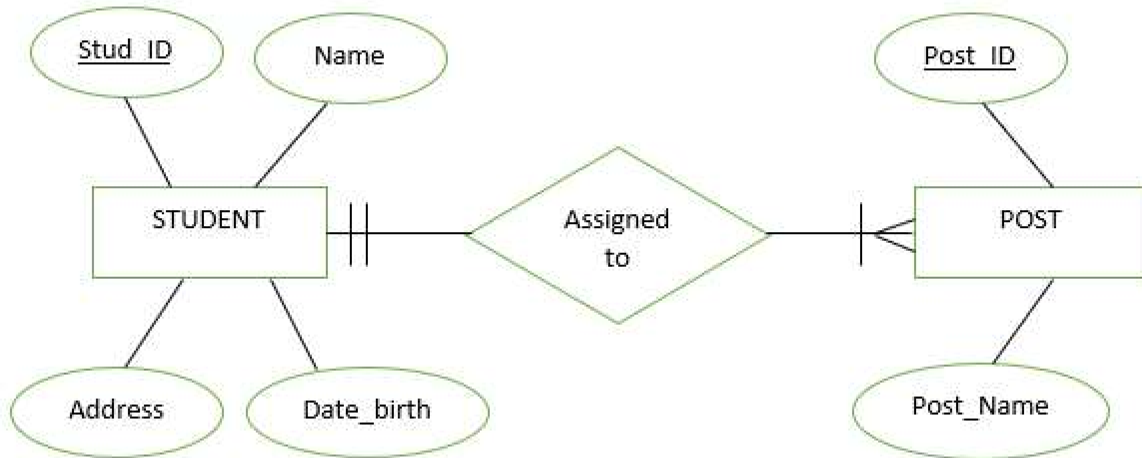
4. A lecturer teaches one course at most, means a lecturer teaches one course or he does not teach any course. A course is taught by one lecturer, means a course is taught by exactly one lecturer.

5. A purchase order may be for many products, means a purchase order contains one or many products. A product may appear on many purchase orders, means a product may appear in many orders and it may not appear in any order at all.



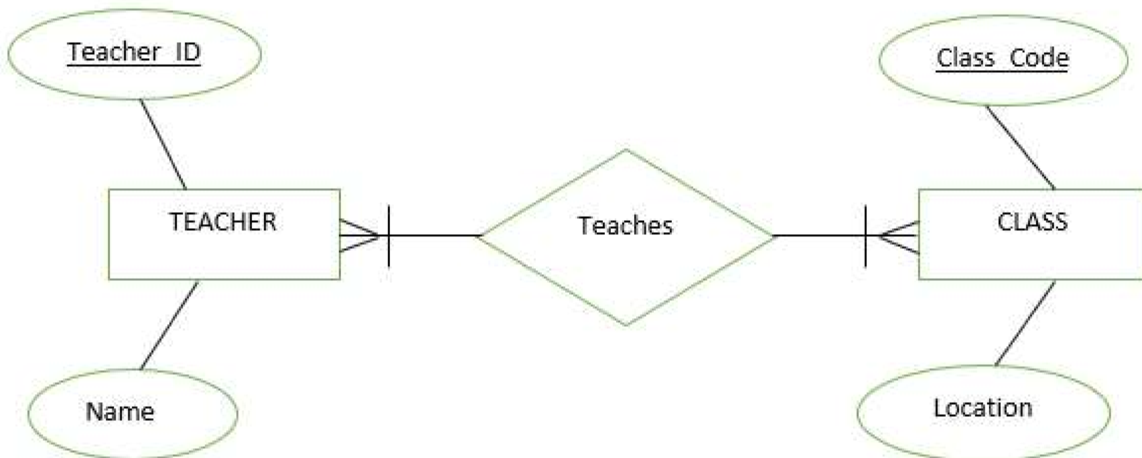
Example 03: Draw an ER Diagram for the following Scenario:

In a school, a student may be assigned to one or more posts like perfect, monitor or chairman. A post must be assigned to exactly one student. A student is identified with student ID, name, address and date of birth. A post is identified with post ID and name.



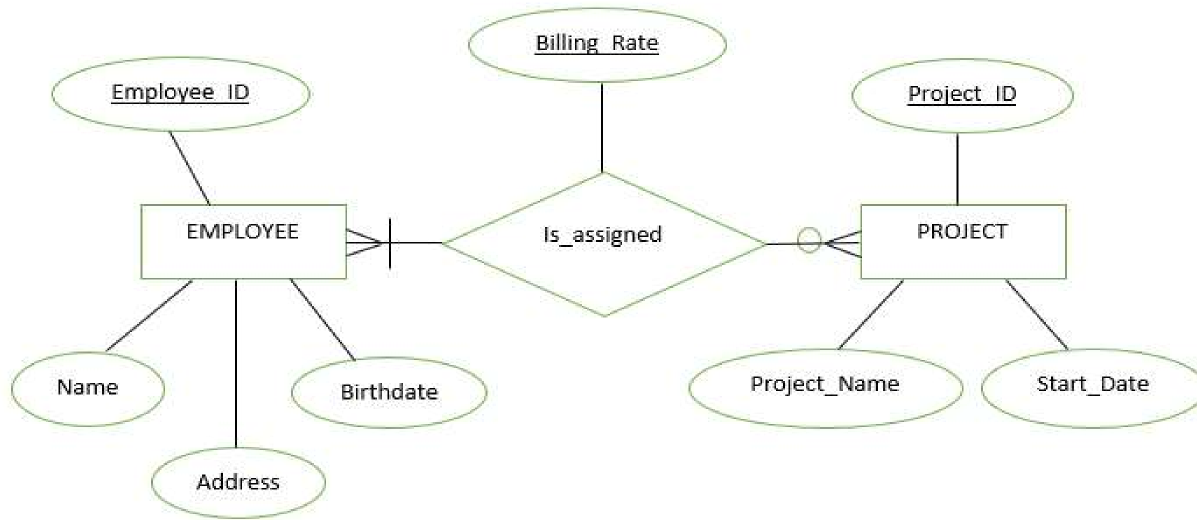
Example 04: Draw an ER Diagram for the following Scenario:

In a school, a teacher teaches one or more classes. Each class is taught by one or more teachers. A teacher is identified with teachers ID and name. A class is identified with class code and location.



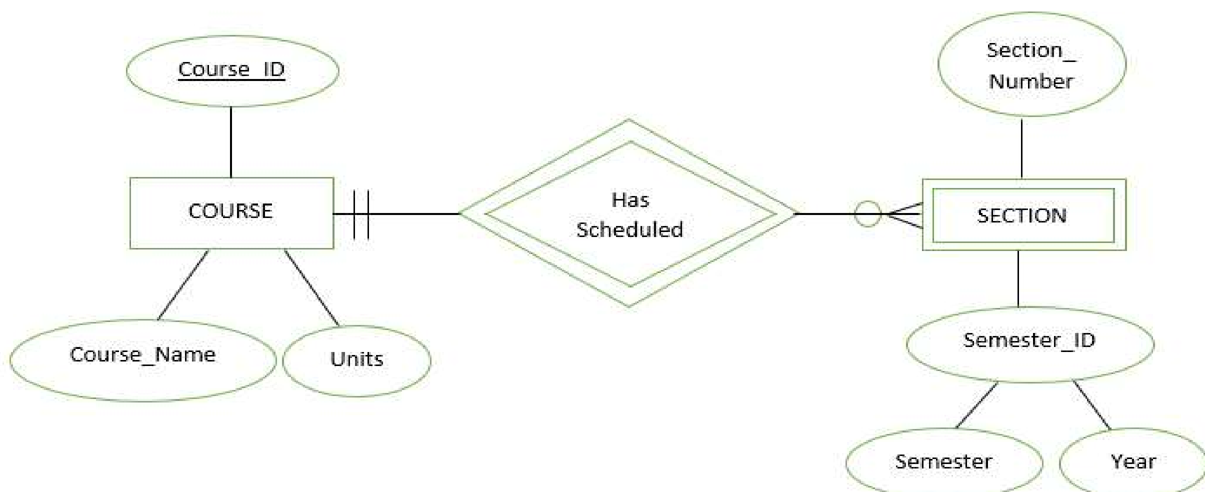
Example 05: Draw an ER Diagram for the following Scenario:

A company has a number of employees. The attributes of EMPLOYEE include Employee_ID (Identifier), Name, Address and Birthdate. The company also has several projects. The attributes of PROJECT are Project_ID (Identifier), Project_Name and Start_Date. Each employee may be assigned to one or more projects or may not be assigned to a project. A project must have at least one employee assigned and may have any number of employees assigned. An employee billing rate may vary by project and that company wishes to record the applicable billing rate (Billing_Rate) for each employee when assigned to a particular project.



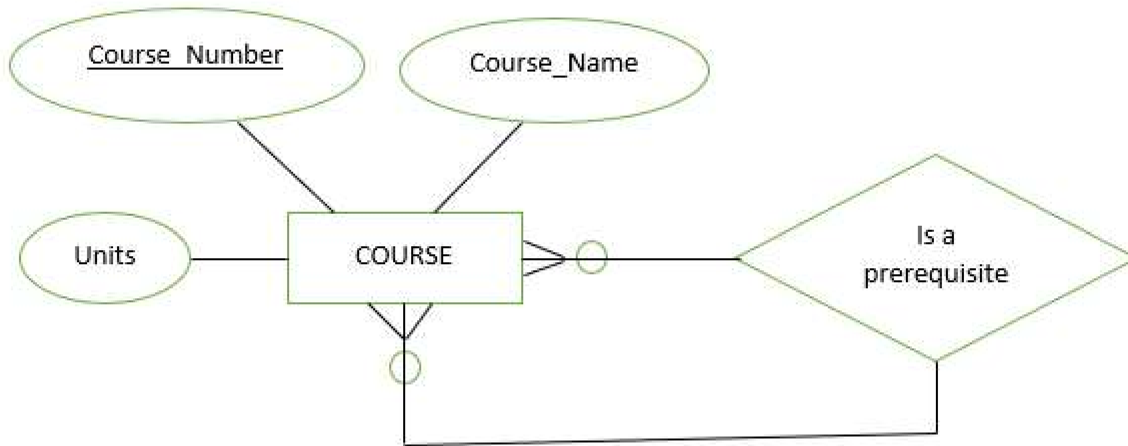
Example 06: Draw an ER Diagram for the following Scenario:

A college course may have one or more scheduled sections or may not have a scheduled section. Attributes of COURSE include Course_ID, Course_ Name, and Units. Attributes of SECTION include Section_Number and Semester_ID. Semester_ID is composed of two parts: Semester and Year. Each SECTION has scheduled by exactly one COURSE.
Note: COURSE is a Strong Entity Type and SECTION is a Weak Entity Type.



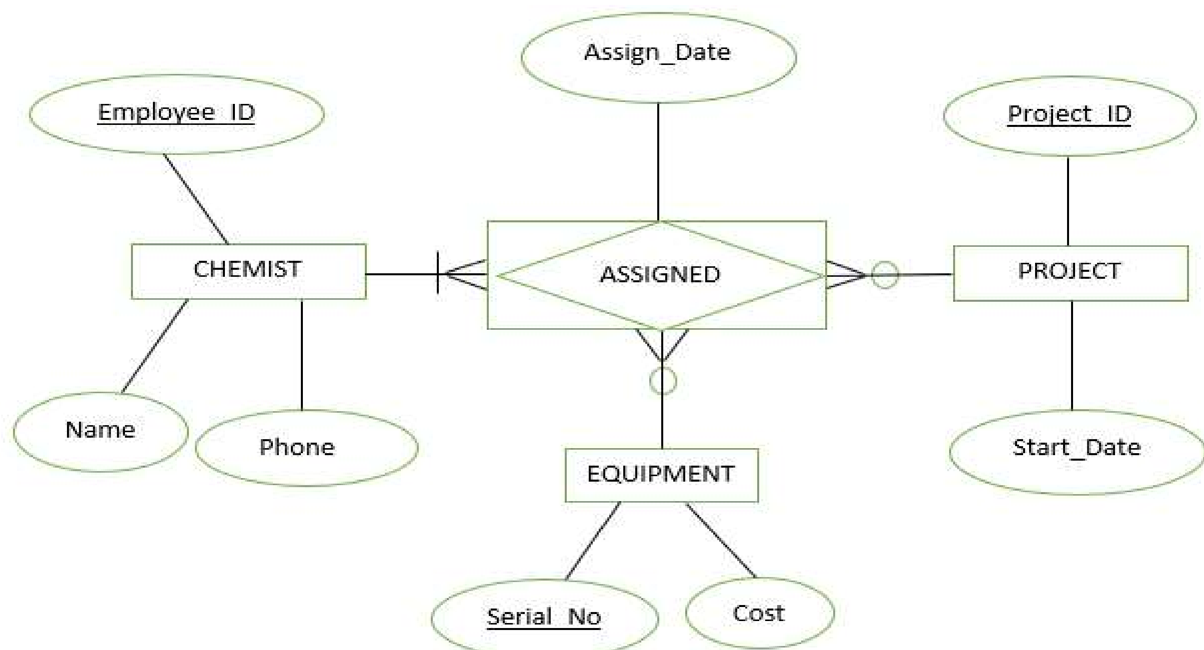
Example 07: Draw an ER Diagram for the following Scenario:

A university has a large number of courses in a catalog. Attributes of COURSE include Course_Number (identifier), Course_Name and Units. Each course may have one or more courses as prerequisites or may have no prerequisites. Similarly, a particular course may be a prerequisite for any number of courses or may not be prerequisite for any other course.



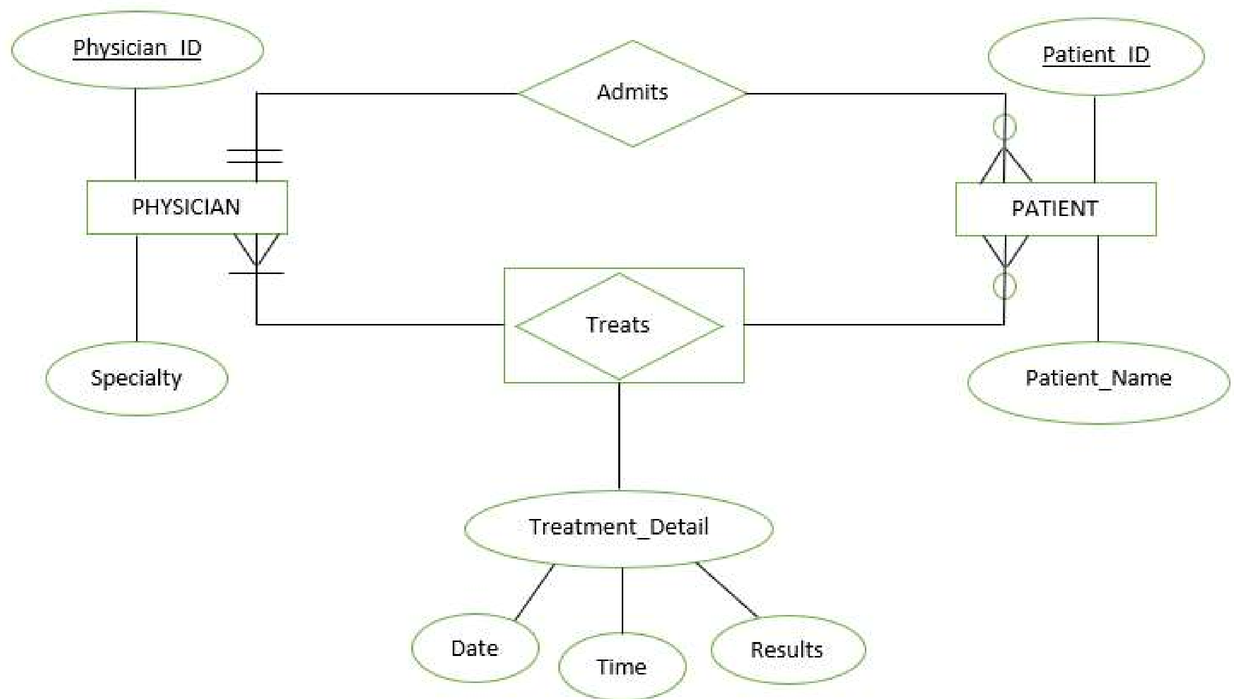
Example 08: Draw an ER Diagram for the following Scenario:

A laboratory has several chemists who work on one or more projects. Chemists also may use certain kinds of equipment on each project. Attributes of CHEMIST include Employee_ID (identifier), Name and Phone_No. Attributes of PROJECT include Project_ID (identifier), and Start_Date. Attributes of EQUIPMENT include Serial_No and Cost. The organization wishes to record Assign_Date i.e., the date when an equipment item was assigned to a particular chemist working on a specific project. A chemist must be assigned to at least one project and one equipment item. An equipment item need not be assigned either a chemist or a project and also a given project need not be assigned either a chemist or an equipment item.



Example 09: Draw an ER Diagram for the following Scenario:

A hospital has a large number of registered physicians. Attributes of PHYSICIAN include Physician_ID (identifier) and Specialty. Patients are admitted to the hospital by physicians. Attributes of PATIENT include Patient_ID (identifier) and Patient_Name. any admitted patient must have exactly one admitting physician. A physician may admit any number of patients. Once admitted, a given patient must be treated by at least one physician. A particular physician may treat any number of patients or may not treat any patients. Whenever a patient is treated by a physician, the hospital records the details of the treatment (Treatment_Detail). Components of Treatment_Detail include Date, Time, and Results.



Week No. 08 & 09: Normalization

Normalization:
Normalization is basically a process of efficiently organizing data in a database. There are **two goals** of the **Normalization** process:

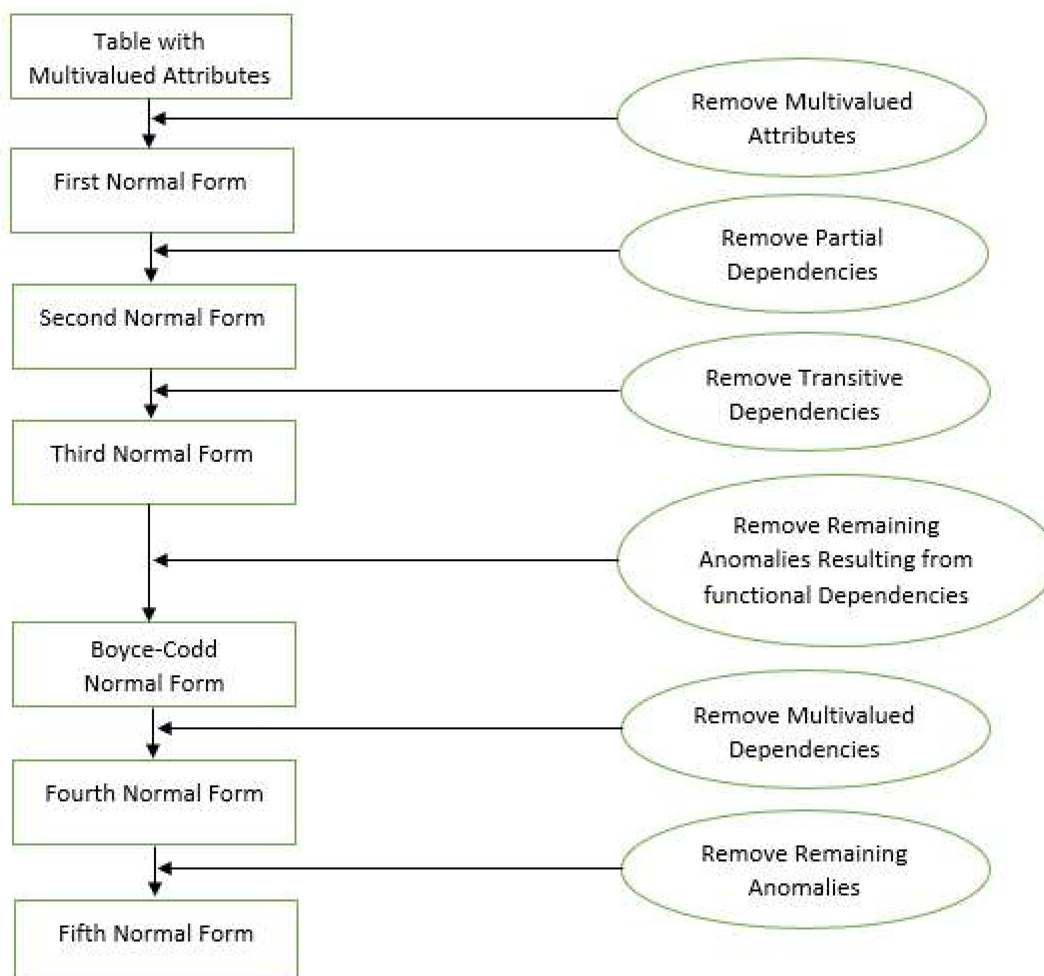
- **Eliminate Redundant Data** (for example, storing the same data in more than one table) and
- **Ensure Data Dependencies** (only storing related data in a table).

Both of these are **worthy goals** as they reduce the **amount** of **space** a database consumes and ensure that **data** is **logically** stored.

Normalization is the **process** of **decomposing** unsatisfactory "**bad**" relations by **breaking up** their **attributes** into **smaller** relations.

Normalization is a **database design technique** which organizes **tables** in a **manner** that reduces **redundancy** and **dependency** of **data**. It divides **larger tables** to **smaller tables** and **links** them using **relationships**.

Steps in Normalization:



Functional Dependency:
Normalization is **based** on the **concept** of **Functional Dependency**. A **Functional Dependency** is a **relationship** between **attributes**. It means that if the **value** of **one attribute** is **known**, it is possible to **obtain** the **value** of **another attribute**. **Suppose** there is a relation **STUDENT** with following fields:

STUDENT (RegistrationNo, StudentName, Class, Email)

If **value** of **RegistrationNo** is **known**, it is possible to **obtain** the **value** of **StudentName**. It means that **StudentName** is **functionally dependent** on **RegistrationNo**.
Functional Dependency is written as **follows**:

RegistrationNo ⟶ StudentName

The above **expression** is **read** as "**RegistrationNo** determines **StudentName**" or "**StudentName** is **functionally dependent** on **RegistrationNo**". The **attribute** on the **left side** is called **determinant**.

A **constraint** between **two attributes** in which the **value** of **one attribute** is **determined** by the **value** of **another attribute**. i.e. For **any** relation **R**, attribute **B** is **functionally dependent** on attribute **A**, if for **every** valid **instance** of **A**, that **value** of **A** uniquely **determines** the **value** of **B** and is **represented** as A ⟶ B.

An **attribute** may be **functionally dependent** on a **single attribute** or a **collection** of **attributes** such **as**:

Std_id, Course_Title ⟶ Date Completed

Other examples:

SSN ⟶ Name, address, DOB

VIN ⟶ Make, Model, Color

ISBN ⟶ Title, Author, Publisher

Determinant: An **attribute** in a **relation** that **uniquely** determine the **value** of **another attribute**. The **attribute**(s) on the **left-hand side** of the **arrow** is/are called **Determinant** e.g., in the above example.

Std_id, Course_Title
SSN
VIN
ISBN

Full Functional Dependency: A **dependency** in which all the **non key attributes** are **fully functionally dependent** on the **Primary Key**.

Partial Dependency: A **Functional Dependency** in which one **or** more **non key attributes** are **functionally dependent** on a **part** (but not all) of the **Primary Key**. For **Partial Dependency**, a **Composite Key** must be **there**. It could be **possible** when there is a **Composite Key**.

Transitive Dependency: A **Functional Dependency** in which **one attribute** functionally determines a **second**, which **functionally** determines a **third**. **Transitive Dependency** occurs when **one non-key attribute** determines **another non-key attribute**.

First Normal Form (1NF):

A **relation** is in **First Normal form** if and only if **every attribute** is **single valued** for **each tuple**. This **means** that **each attribute** in **each row**, or **each cell** of the **table**, contains only **one value**. **No repeating groups** are **allowed**. A **repeating group** is a **set** of one **or** more **data items** that may occur a **variable number** of **times** in a **tuple**. The **value** in **each attribute** should be **atomic** and **every tuple** should be **unique**. There is **no multivalued** (repeating groups) in the **Relation**.

Example No. 01: There is a **Relation** of **Student**.

STD (stId, stName, stAdr, prName, bkId)

Primary Key

stId	**stName**	**stAdr**	**prName**	**bkId**
S1020	Sohail Dar	I-8 Islamabad	MCS	B00129
S1038	Shoaib Ali	G-6 Islamabad	BCS	B00327
S1015	Tahira Ejaz	L Rukh Wah	MCS	B08945, B06352
S1018	Arif Zia	E-8, Islamabad.	BIT	B08474

Now in this **table** there is **no unique value** for **every tuple**, like for **S1015** there are **two values** for **bookId**. So, to bring it in the **First Normal form**.

Primary Key

stId	**stName**	**stAdr**	**prName**	**bkId**
S1020	Sohail Dar	I-8 Islamabad	MCS	B00129
S1038	Shoaib Ali	G-6, Islamabad	BCS	B00327
S1015	Tahira Ejaz	L Rukh Wah	MCS	B08945
S1015	Tahira Ejaz	L Rukh Wah	MCS	B06352
S1018	Arif Zia	E-8, Islamabad.	BIT	B08474

Now this **table** is in **First Normal Form** and for **every tuple** there is a **unique value**.

Example No. 02:

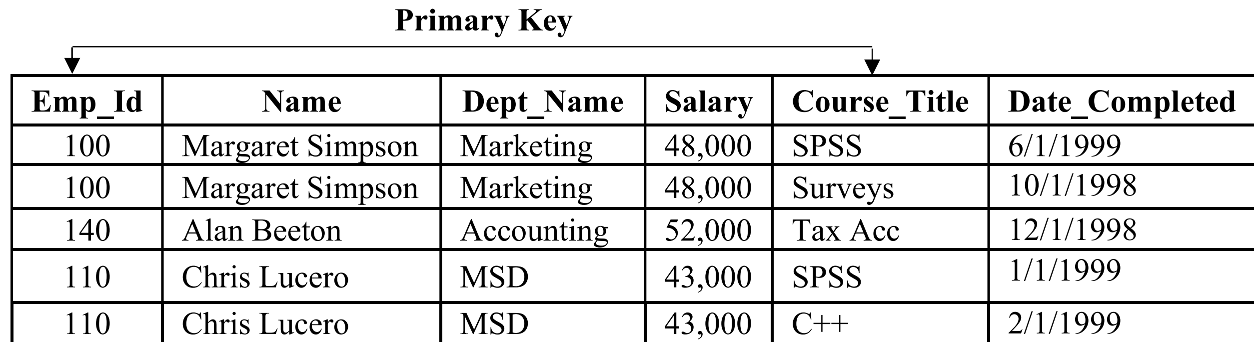
Unnormalized Form (UNF):
A **relation** containing one **or** more **repeating groups**.

Primary Key

Emp_Id	**Name**	**Dept_Name**	**Salary**	**Course_Title**	**Date_Completed**
100	Margaret Simpson	Marketing	48,000	SPSS Surveys	6/1/1999 10/1/1998
140	Alan Beeton	Accounting	52,000	Tax Acc	12/1/1998
110	Chris Lucero	MSD	43,000	SPSS C++	1/1/1999 2/1/1999

First Normal Form (1NF):

In **First Normal Form**, there is **no multi-valued attributes** i.e., **intersection** of **each row** and **column** contain **one** and only **one value**. Every **attribute value** is **atomic** (or indivisible).



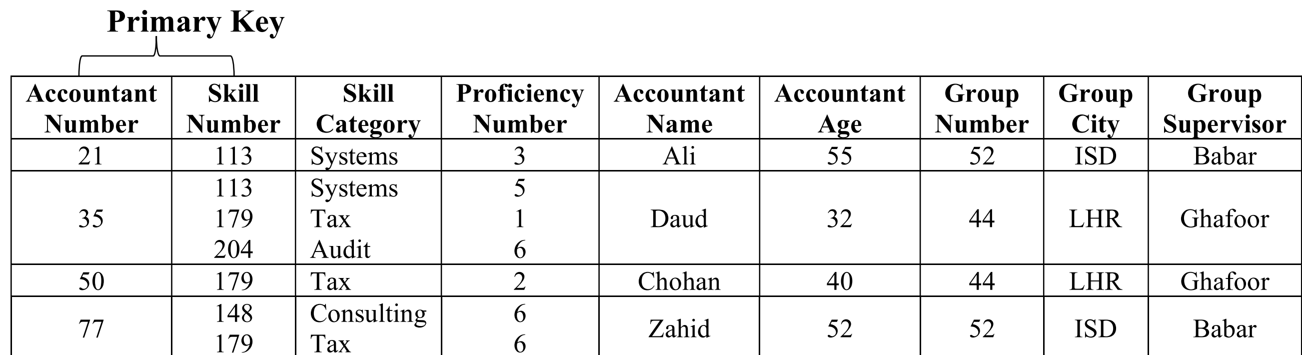
Primary Key

Emp_Id	Name	Dept_Name	Salary	Course_Title	Date_Completed
100	Margaret Simpson	Marketing	48,000	SPSS	6/1/1999
100	Margaret Simpson	Marketing	48,000	Surveys	10/1/1998
140	Alan Beeton	Accounting	52,000	Tax Acc	12/1/1998
110	Chris Lucero	MSD	43,000	SPSS	1/1/1999
110	Chris Lucero	MSD	43,000	C++	2/1/1999

Figure: First Normal Form (1NF)

Example No. 03:

Unnormalized Form (UNF):



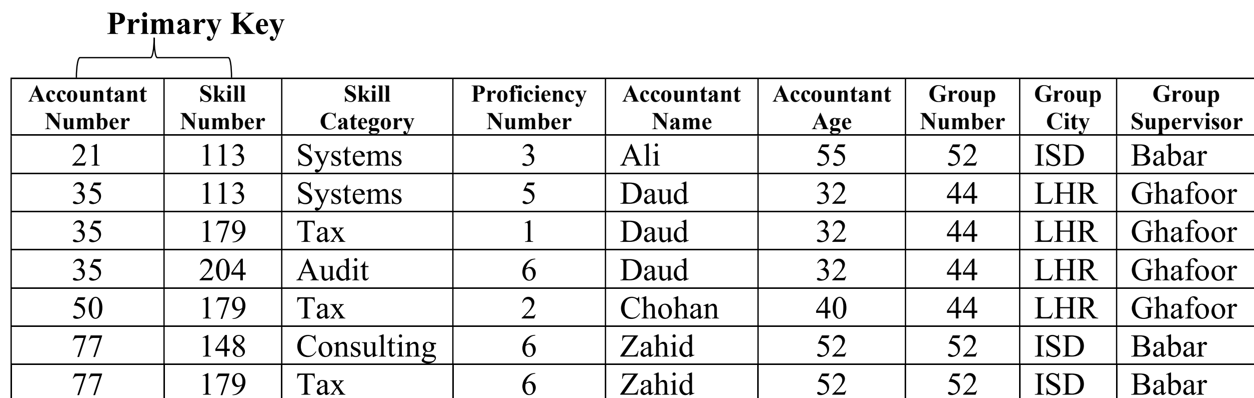
Primary Key

Accountant Number	Skill Number	Skill Category	Proficiency Number	Accountant Name	Accountant Age	Group Number	Group City	Group Supervisor
21	113	Systems	3	Ali	55	52	ISD	Babar
35	113 179 204	Systems Tax Audit	5 1 6	Daud	32	44	LHR	Ghafoor
50	179	Tax	2	Chohan	40	44	LHR	Ghafoor
77	148 179	Consulting Tax	6 6	Zahid	52	52	ISD	Babar

Figure: Un-normalize Relation

The **above relation** is **un-normalized** because it contains **repeating groups** of **three attributes**: Skill Number, Skill Category and Proficiency Number. **All** these **fields** contain **more than one value**. In order to convert this into **Normal Form**, these **repeating groups** should be **removed**.

The **following relation** is in **First Normal Form**:



Primary Key

Accountant Number	Skill Number	Skill Category	Proficiency Number	Accountant Name	Accountant Age	Group Number	Group City	Group Supervisor
21	113	Systems	3	Ali	55	52	ISD	Babar
35	113	Systems	5	Daud	32	44	LHR	Ghafoor
35	179	Tax	1	Daud	32	44	LHR	Ghafoor
35	204	Audit	6	Daud	32	44	LHR	Ghafoor
50	179	Tax	2	Chohan	40	44	LHR	Ghafoor
77	148	Consulting	6	Zahid	52	52	ISD	Babar
77	179	Tax	6	Zahid	52	52	ISD	Babar

Figure: First Normal Form (1NF)

Second Normal Form (2NF):
A **relation** is in **Second Normal Form** (2NF) if and only if it is in **First Normal Form** and all the **non key attributes** are **fully functionally dependent** on the **whole key**. It **means** that **none** of **non-key attributes** are **related** to a **part of key**. Clearly, if a **relation** is in **1NF** and the **key** consists of a **single attribute**, the **relation** is **automatically** in **2NF**. The **concern** about **2NF** is when the **key** is **composite**.

Second Normal Form (2NF) addresses the **concept** of **removing duplicative data**. It removes **subsets** of **data** that **apply** to **multiple rows** of a **table** and **place** them in separate tables. It creates **relationships** between these **new tables** and their **predecessors** through the use of **Foreign Keys**.

Example No. 01: Consider the following **Relation**.

Primary Key

Accountant Number	Skill Number	Skill Category	Proficiency Number	Accountant Name	Accountant Age	Group Number	Group City	Group Supervisor
21	113	Systems	3	Ali	55	52	ISD	Babar
35	113	Systems	5	Daud	32	44	LHR	Ghafoor
35	179	Tax	1	Daud	32	44	LHR	Ghafoor
35	204	Audit	6	Daud	32	44	LHR	Ghafoor
50	179	Tax	2	Chohan	40	44	LHR	Ghafoor
77	148	Consulting	6	Zahid	52	52	ISD	Babar
77	179	Tax	6	Zahid	52	52	ISD	Babar

Figure: First Normal Form (1NF)

The **above relation** in **1NF** has **some attributes** which are **not depending** on the **whole primary key**. For **example**, Accountant Name, Accountant Age and group information is determined by **Accountant Number** and is **not dependent** on **Skill Number**. The **following relation** can be created in which **all attributes** are **fully dependent** on **Primary Key** (Accountant Number).

Primary Key

Accountant Number	Accountant Name	Accountant Age	Group Number	Group City	Group Supervisor
21	Ali	55	52	ISD	Babar
35	Daud	32	44	LHR	Ghafoor
50	Chohan	40	44	LHR	Ghafoor
77	Zahid	52	52	ISD	Babar

Figure: Accountant Table in 2NF

Similarly, another relation **Skill** can be created in which **all fields** are **fully dependent** on the **Primary Key** as **follows**:

Primary Key

Skill Number	Skill Category
113	Systems
179	Tax
204	Audit
148	Consulting

Figure: Skill Table in 2NF

The **attribute Proficiency** in **1NF relation** was **fully dependent** on the **whole Primary Key**. The **Proficiency** requires to know the **Accountant Number** and **Skill Number**. The **third relation** will be created as **follows**:

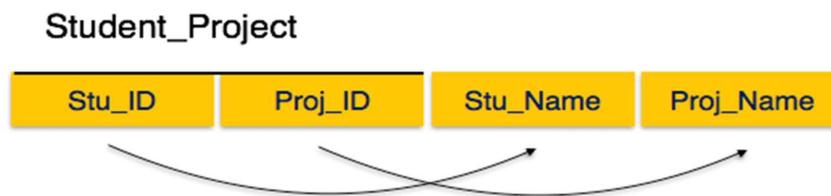
Primary Key (Accountant Number, Skill Number)

Accountant Number	Skill Number	Proficiency Number
21	113	3
35	113	5
35	179	1
35	204	6
50	179	2
77	148	6
77	179	6

Figure: Proficiency Table in 2NF

There are **three relations** in **Second Normal Form** (2NF). The **attributes** of **all relations** are **fully dependent** on the **Primary Keys**.

Example No. 02: Consider the following relation.



Here in **Student_Project** relation that the **Prime Key attributes** are **Stu_ID** and **Proj_ID**. **According** to the **rule, non-key attributes**, i.e., **Stu_Name** and **Proj_Name** must be **dependent** upon **both** and **not** on **any** of the **Prime Key attribute individually**. But **Stu_Name** can be **identified** by **Stu_ID** and **Proj_Name** can be **identified** by **Proj_ID independently**. This is called **Partial Dependency**, which is **not allowed** in **Second Normal Form**.
Decomposed **Student_Project relation** into **two separate relations**.

Student

Stu_ID	Stu_Name	Proj_ID

Project

Proj_ID	Proj_Name

Student (Stu_ID, Stu_Name, Proj_ID)

Project (Proj_ID, Proj_Name)

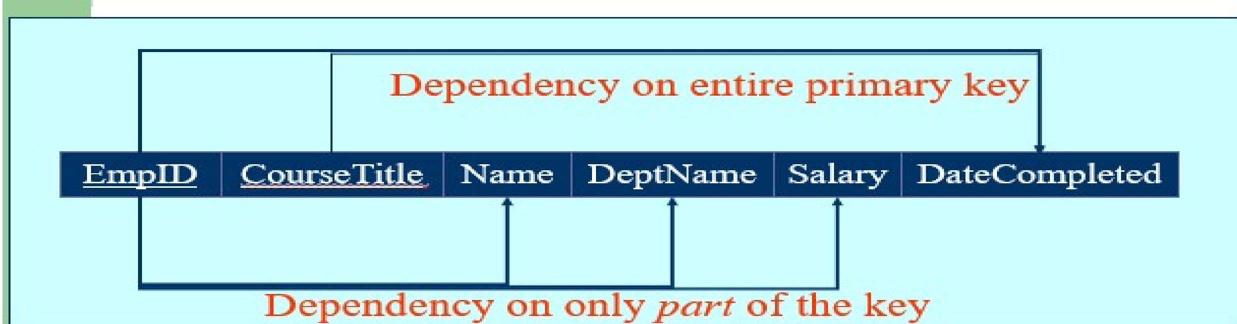
Now these **two relations/tables** are in **Second Normal Form**.

Example No. 03:
Relation **EMPLOYEE** is **NOT** in **2nd Normal Form** (Name, Department Name, and Salary is only **dependent** on **Emp_Id**).

Primary Key

Emp_Id	Name	Dept_Name	Salary	Course_Title	Date_Completed
100	Margaret Simpson	Marketing	48,000	SPSS	6/1/1999
100	Margaret Simpson	Marketing	48,000	Surveys	10/1/1998
140	Alan Beeton	Accounting	52,000	Tax Acc	12/1/1998
110	Chris Lucero	MSD	43,000	SPSS	1/1/1999
110	Chris Lucero	MSD	43,000	C++	2/1/1999

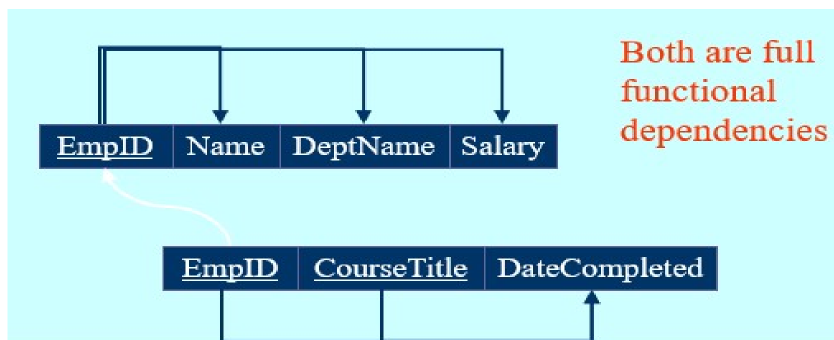
Functional Dependencies in EMPLOYEE table:



EmpID, CourseTitle → DateCompleted
EmpID → Name, DeptName, Salary

Therefore, NOT in 2nd Normal Form!!

Decomposed **EMPLOYEE** relation into **two separate relations**.



Emp_ID (Emp_Id, Name, Dept_Name, Salary)

Course_Title (Emp_Id, Course_Title, Date_Completed)

Primary Key

Emp_Id	Name	Dept_Name	Salary
100	Margaret Simpson	Marketing	48,000
140	Alan Beeton	Accounting	52,000
110	Chris Lucero	MSD	43,000

Figure: Emp_ID Table in 2NF

Primary Key

Emp_Id	Course_Title	Date_Completed
100	SPSS	6/1/1999
100	Surveys	10/1/1998
140	Tax Acc	12/1/1998
110	SPSS	1/1/1999
110	C++	2/1/1999

Figure: Course_Title Table in 2NF

Now these **two relations/tables** are in **Second Normal Form**.

Third Normal Form:

A **relational table** is in **Third Normal Form** (3NF) if it is **already** in **2NF** and if no **non-key attribute** is **dependent** on **another non-key attribute**. It **means** that all **non-key attribute** is **non-transitively dependent** upon its **Primary Key**. There should be no **transitive dependency** in a **relation**. In **other words**, all **non-key attributes** are **functionally dependent** only upon the **Primary Key**.

For **Third Normal Form**, concentrate on **relations** with **one Candidate Key** and **eliminate Transitive Dependencies**. **Transitive Dependency** occurs when **one non-key attribute** determines **another non-key attribute**.

Example No. 01:

Primary Key

Accountant Number	Accountant Name	Accountant Age	Group Number	Group City	Group Supervisor
21	Ali	55	52	ISD	Babar
35	Daud	32	44	LHR	Ghafoor
50	Chohan	40	44	LHR	Ghafoor
77	Zahid	52	52	ISD	Babar

Figure: Accountant Table in 2NF

The **Accountant Table** in **2NF** contains **some attributes** which are **depending** on **non-key attributes**. For **example**, **Group City** and **Group Supervisor** are **depending** on a **non-key** field **Group Number**. A **new relation** can be created as **follows**:

Primary Key

Accountant Number	Accountant Name	Accountant Age	Group Number
21	Ali	55	52
35	Daud	32	44
50	Chohan	40	44
77	Zahid	52	52

Figure: Accountant Table in 3NF

The **Second Table** created from the **Accountant Table** in **2NF** as **follows**:

Primary Key

Group Number	Group City	Group Supervisor
52	ISD	Babar
44	LHR	Ghafoor

Figure: Group Table in 3NF

Both **Accountant Table** and **Group Table** contain the attribute **Group Number**. This **attribute** is used to **join** both **tables**.

The **Skill Table** and **Proficiency Table** both in **2NF** contains **no attribute**, which is **depending** on a **non-key attribute**. They are **already** in **Third Normal Form** and will be used **without** any **further change**.

Primary Key

Skill Number	Skill Category
113	Systems
179	Tax
204	Audit
148	Consulting

Figure: Skill Table in 3NF

Primary Key

Accountant Number	Skill Number	Proficiency Number
21	113	3
35	113	5
35	179	1
35	204	6
50	179	2
77	148	6
77	179	6

Figure: Proficiency Table in 3NF

Example No. 02:

STUDENT

stId	stName	stAdr	prName	prCrdts
S1020	Sohail Dar	I-8 Islamabad	MCS	64
S1038	Shoaib Ali	G-6, Islamabad	BCS	132
S1015	Tahira Ejaz	L Rukh Wah	MCS	64
S1018	Arif Zia	E-8, Islamabad	BIT	134

Transitive Dependency:

STD (stId, stName, stAdr, prName, prCrdts)

stId → stName, stAdr, prName, prCrdts
prName → prCrdts

Now here the **STUDENT** table is in **Second Normal Form**. As there is **no Partial Dependency** of any attributes here. The **key** is **student ID**. The **problem** is of **Transitive Dependency** in which a **non-key attribute** can be determined by a **non-key attribute**. Like here the **program credits** can be determined by **program name**, which is **not** in **3NF**.
Decomposed **STUDENT** relation into **two separate relations**.

STD (stId, stName, stAdr, prName)

PROGRAM (prName, prCrdts)

stId	stName	stAdr	prName
S1020	Sohail Dar	I-8 Islamabad	MCS
S1038	Shoaib Ali	G-6, Islamabad	BCS
S1015	Tahira Ejaz	L Rukh Wah	MCS
S1018	Arif Zia	E-8, Islamabad	BIT

Figure: STD Table in 3NF

prName	prCrdts
MCS	64
BCS	132
MCS	64
BIT	134

Figure: Program Table in 3NF

Now these two **relations/tables** are in **Third Normal Form**.

Example No. 03:

In the above **Student_detail** relation, **Stu_ID** is the **key** and only **Prime Key attribute**. **City** can be **identified** by **Zip**. **Zip** and **City** both are **non-key attributes**. so there exists **Transitive Dependency**.

Student_Detail (Stu_ID, Stu_Name, City, Zip)

Stu_ID ———→ Stu_Name, Zip

Zip ———→ City

To bring this relation into **Third Normal Form**, decomposed the **relation** into two **relations** as **follows**:

Student_Detail (Stu_Name, Zip)

ZipCodes (Zip, City)

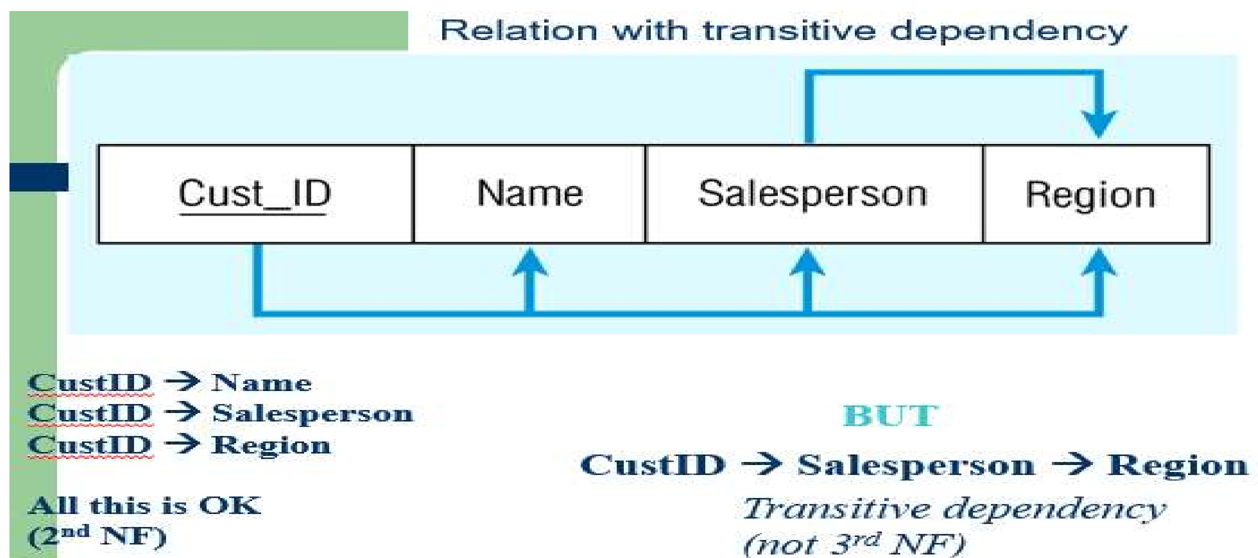
Now these **two relations**/**tables** are in **Third Normal Form**.

Example No. 04:

Sales relation is **NOT** in **3rd Normal Form** (**Region** is depend on **Salesperson**).

Cust_ID	Name	Salesperson	Region
8023	Anderson	Smith	South
9167	Bancroft	Hicks	West
7924	Hobbs	Smith	South
6837	Tucker	Hernandez	East
8596	Eckersley	Hicks	West
7018	Arnold	Faulb	North

Figure: SALES relation



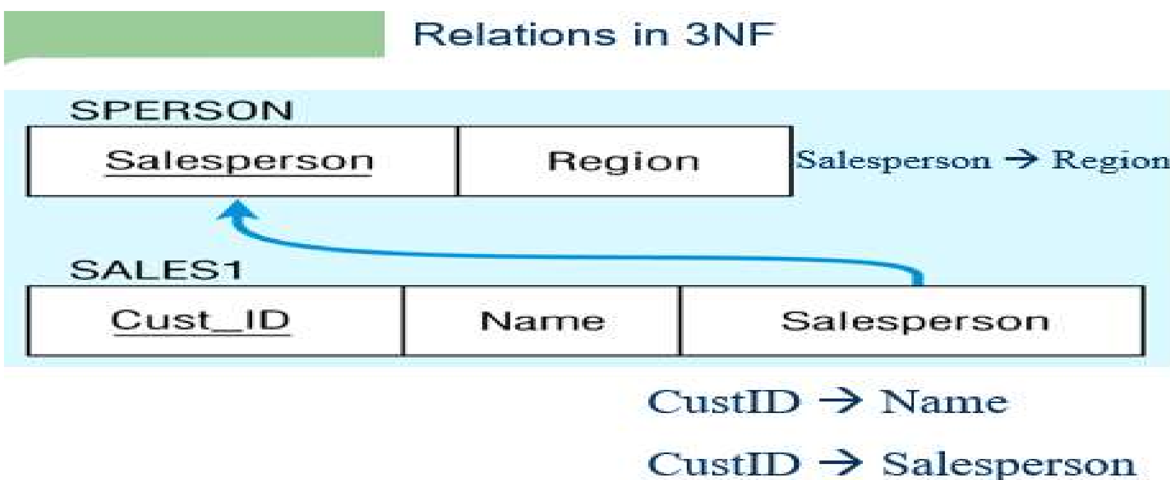
Removing a Transitive Dependency:
Decomposing the SALES relation into the following two relations.

Cust_ID	Name	Salesperson
8023	Anderson	Smith
9167	Bancroft	Hicks
7924	Hobbs	Smith
6837	Tucker	Hernandez
8596	Eckersley	Hicks
7018	Arnold	Faulb

Figure: SALES1 relation

Salesperson	Region
Smith	South
Hicks	West
Hernandez	East
Faulb	North

Figure: SPERSON relation



Boyce - Codd Normal Form:
A **relation** is in **Boyce-Codd Normal Form** if and only if every **determinant** is a **Candidate Key**. It can be **checked** by **identifying** all **determinants** and then making sure that **all** these **determinants** are **Candidate Keys**. **BCNF** is a **stronger form** of **Third Normal Form**. A **relation** in **BCNF** is also in **Third Normal Form**. But a **relation** in **3NF** may not be in **BCNF**.

Example No. 01:

<u>SID</u>	Major	<u>Advisor</u>	GPA
1	DB	Abid	3.5
2	C++	Arshad sb	3.6

Figure: Student relation in 3rd form.

The above STUDENT relation satisfies 2nd and 3rd form of Normalization because there are **no non-key attributes** that are **dependent** on a **subset** of the **Primary Key** and also there are **no transitive dependencies** but not in BCNF because every determinant is not a Candidate Key.

In order to **convert** this **relation** to **BCNF**, **all functional dependencies** must be removed which have a **determinant** that is not a **Candidate Key**.

The **result** is as **follows**:

<u>SID</u>	Advisor	<u>Major</u>	GPA
1	Abid	DB	3.5
2	Arshad sb	C++	3.6

Figure: Student relation in BCNF (Partial Dependency occurs).

But here Partial Dependency occurs between major and advisor.

<u>SID</u>	<u>Major</u>	GPA
1	DB	3.5
2	C++	3.6

Figure: Student relation in BCNF.

<u>Major</u>	Advisor
DB	Abid
C++	Arshad sb

Figure: Major relation in BCNF.

Example No. 02:

Primary Key

<u>ProjectID</u>	PartID	Quantity Used	<u>PartName</u>
101	P01	20	CD-R
112	P05	6	Zip disk
194	P01	12	CD-R
194	P02	1	Box floppy disks
194	P05	3	Zip disk

Figure: Project and Parts relation

There are following dependencies:

(ProjectID, PartID) ⟶ QtyUsed

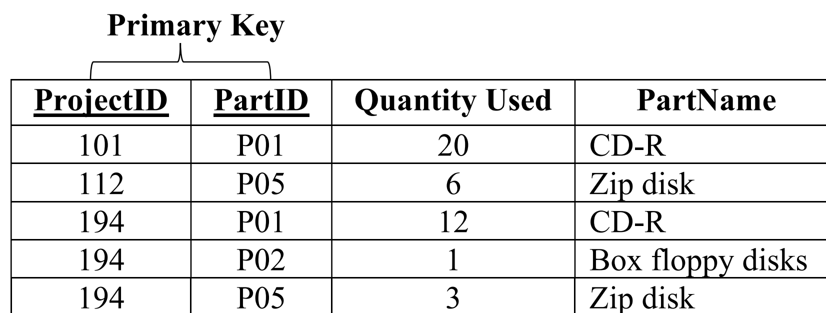
PartID ⟶ PartName

This **relation** satisfies **2NF** because there are **no non-key attributes** that are **dependent** on a **subset** of the **Primary Key**. **PartName** is not a **non-key attribute**, it is **part** of a **Candidate Key**. Therefore, this **relation** is in **2NF**.

There are **no transitive dependencies** so the **relation** is in **3NF**. **PartID** and **PartName** are **ignored** by **3NF** rule because they are **both Key Attributes**.

In order to **convert** this **relation** to **BCNF**, **all functional dependencies** must be removed which have a **determinant** that is not a **Candidate Key**.

The **result** is as **follows**:

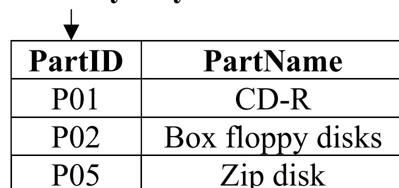


Primary Key

ProjectID	PartID	Quantity Used	PartName
101	P01	20	CD-R
112	P05	6	Zip disk
194	P01	12	CD-R
194	P02	1	Box floppy disks
194	P05	3	Zip disk

Figure: Project and Parts relation in BCNF (Partial Dependency occurs).

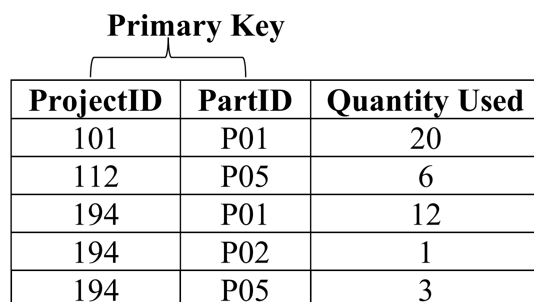
But here Partial Dependency occurs between PartID and PartName.



Primary Key

PartID	PartName
P01	CD-R
P02	Box floppy disks
P05	Zip disk

Figure: Parts relation in BCNF



Primary Key

ProjectID	PartID	Quantity Used
101	P01	20
112	P05	6
194	P01	12
194	P02	1
194	P05	3

Figure: Project relation in BCNF

Transforming of ER-Diagram to Relations:
ER Model represents different things as entities. The connections among different entities are represented by relationships. These entities and relationship can be transformed into relational model. This model can used to design the database.

Converting Regular (Strong) Entities into Relations:
The process of converting entities into relations is very simple. In this process, the **name** of **entity** becomes the **name** of **relation** and **attributes** of **entity** becomes the **fields** of **relation**.

Example:

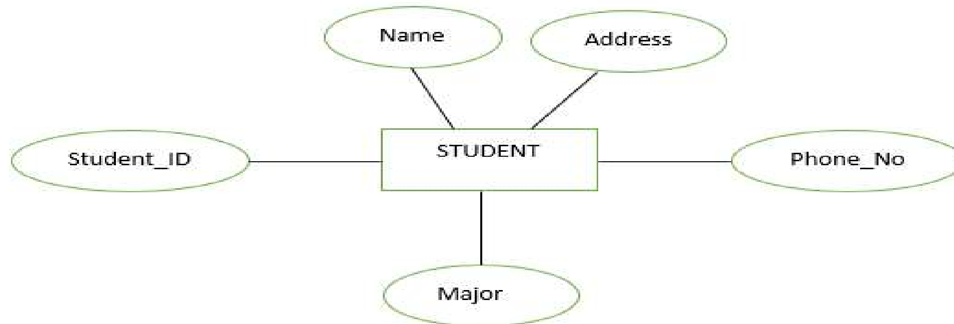


Figure: STUDENT Entity

Student_ID	Name	Address	Phone_No	Major

Figure: STUDENT Entity

Converting Composite Attributes:
If an **entity** contains **composite attributes**, **each part** of the **attribute** is represented by a **separate field** in the **relation**.

Example:

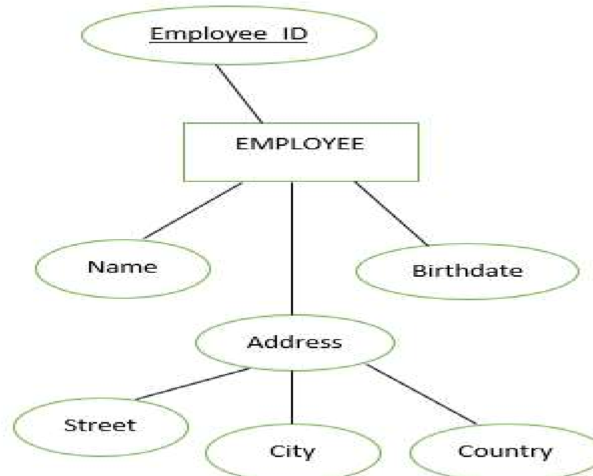


Figure: EMPLOYEE Entity

Employee_ID	Name	Birthdate	Street	City	Country

Figure: EMPLOYEE Relation

Multi-valued Attribute:
The **attribute** is represented in a **separate relation** with a **Foreign Key** taken from the **Superior Entity** if an **entity** contains **Multi-valued Attribute**.

Example:

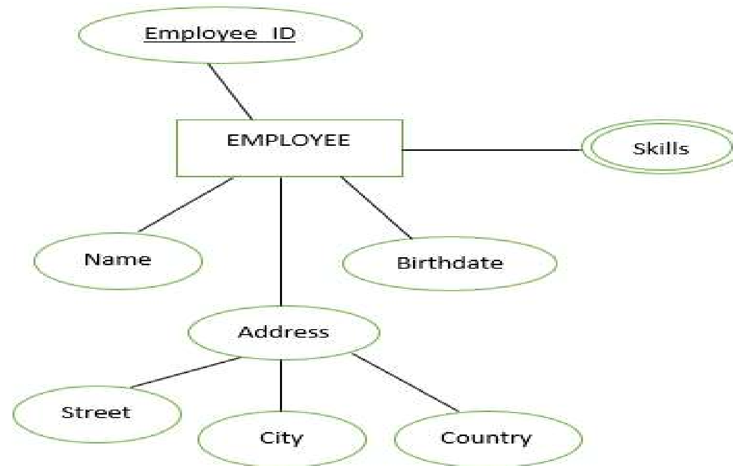


Figure: EMPLOYEE Entity with multi-valued attributes

Employee_ID	Name	Birthdate	Street	City	Country

Figure: EMPLOYEE Relation

Employee_ID	Skills

Figure: SKILLS Relation

The second relation **SKILLS** is using **Employee_ID** and **Skill** as **Composite Key**.

Converting Weak Entities into Relation:

A **Weak Entity** does not exist **independently**. It depends on the existence of another entity known as **Identifying Owner**. When **entities** are converted into **relations**, first of all a **relation** for the **identifying owner** is created. A **separate relation** is created for **each Weak Entity** and **attributes** of **Weak Entity** become the **fields** of the **relation**. The **Weak Entity** relation is connected with the **Identifying Relation**. The **Primary Key** of **Identifying Relation** is used as **Foreign Key** in the **Weak Entity** relation.

Example:

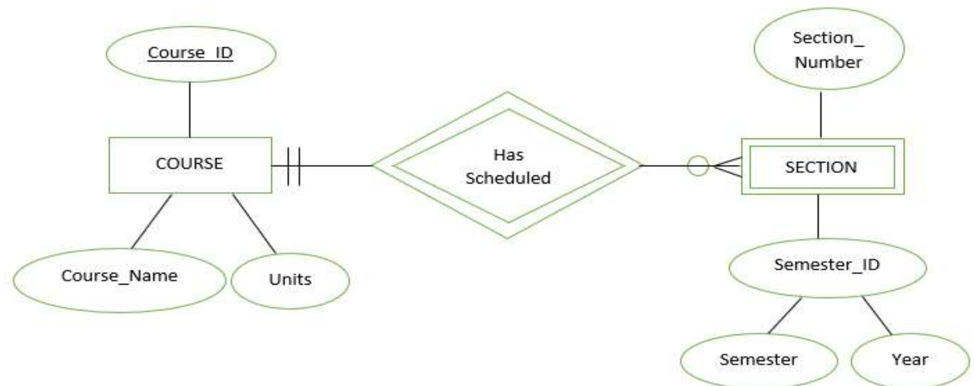


Figure: Weak Entity SECTION

Course_ID	Course_Name	Units

Section_Number	Course_ID	Semester	Year

Figure: COURSE and SECTION relations

Converting Binary Relationships into Relations:

The process of representing Relationships depends upon two things:

1. Degree of Relationship
2. Cardinality of Relationship

Binary One-to-One Relationship:

One-to-One Relationship of **ER Model** is represented in **relations** by performing the following **two steps**:

i. Create a relation for each of the two entity types participating in the relationship.

ii. Include the Primary Key of the first relation as Foreign Key in the second relation.

Example:

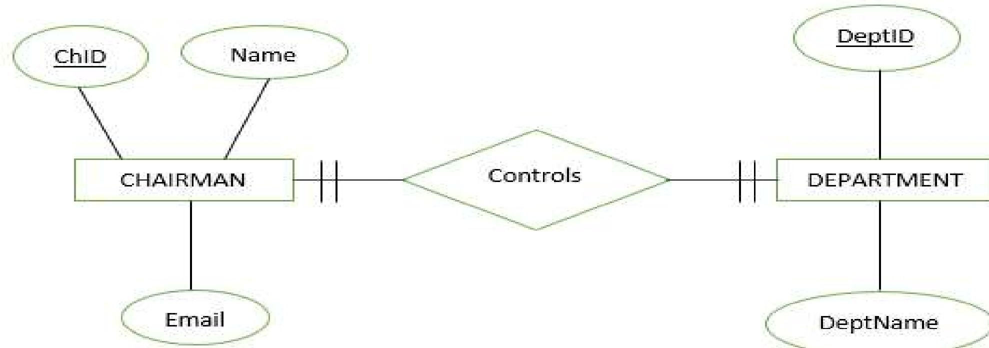


Figure: Binary One-to-One Relationship

ChID	Name	Email

DeptID	DeptName	ChID

Figure: Relations in Binary One-to-One Relationship

Binary One-to-Many Relationship:

One-to-Many relationship of **ER Model** is represented in relations by performing the following two steps:

i. Create a relation for each of the two entity types participating in the relationship.

ii. Include the Primary Key of the entity on One-side of the relationship as a Foreign Key in the relation that is on the Many-side of the relationship.

Example:

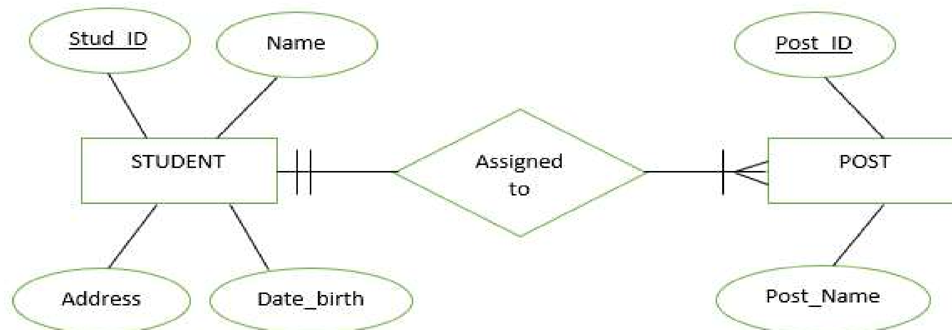


Figure: Binary One-to-Many Relationship

Stud_ID	Name	Date_Birth	Address

Post_ID	Post_Name	Stud_ID

Figure: Relations in Binary One-to-Many Relationship

Binary Many-to-Many Relationship:
Many-to-Many relationship of **ER Model** is represented in relations by **performing** the following **two steps**:

i. Create two relations A and B for each of the two entity types participating in the relationship.

ii. Create another relation C (association relation) that contains Primary Keys of relations A and B as a Foreign Key. These attributes become the Primary of the relation C.

Example:

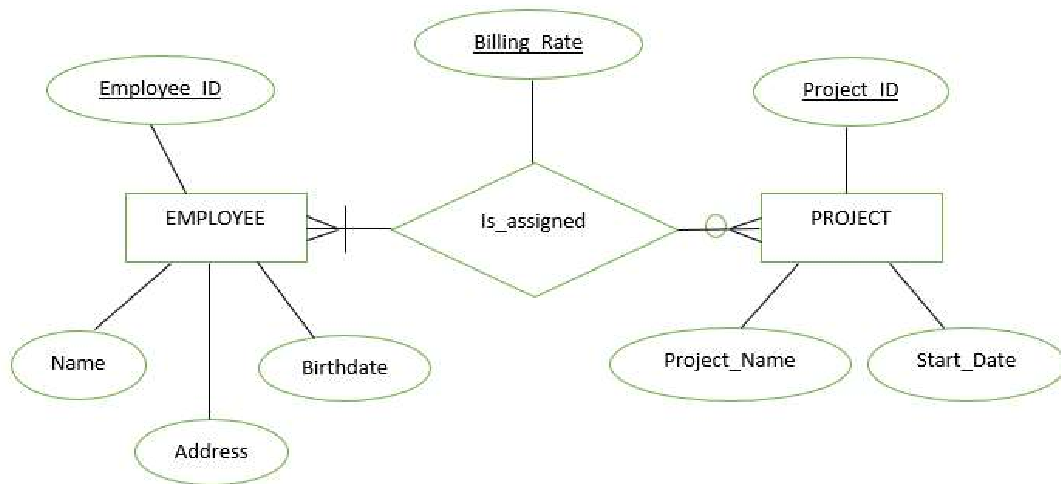


Figure: Binary Many-to-Many Relationship

Employee_ID	Name	Address	Date_Birth

Employee_ID	Project_ID	Billing_Rate

Project_ID	Project_Name	Start_Date

Figure: Relations in Binary Many-to-Many Relationship

Converting Unary Relationships into Relations:
Unary Relationship exists between the **instances** of **same entity types**. It is also known as **Recursive Relationship**. **Unary Many-to-Many relationship** of **ER Model** is represented in **relations** by **performing** the following **two steps**:

i. Create a relation to represent the entity type.

ii. Add another field as Foreign Key in the same relation that references the Primary Key of the relation. The Foreign Key must have the same domain as the Primary Key.

Example:

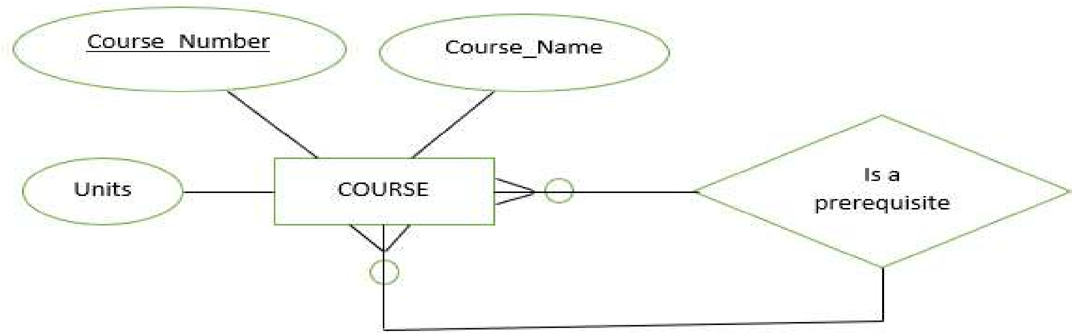


Figure: Unary Many-to-Many Relationship

Course_Number	Course_Name	Units	Prerequisite_Number

Figure: Relation in Unary Many-to-Many Relationship

Converting Ternary Relationships into Relations:

Ternary Relationship exists between the **instances** of **three entity types**. **Ternary Relationship** of **ER Model** is represented in **relations** by **performing** the following **two steps**:

i. Create a relation for each entity type participating in the relationship.
ii. Create an associative relation to represent the link between three entities.

The **Primary Key** of **associative relation** consists of **Primary Keys** of **three entities** in the relationship. The **attributes** of **associative entity type** become **attribute** of the **new relation**.

Example:

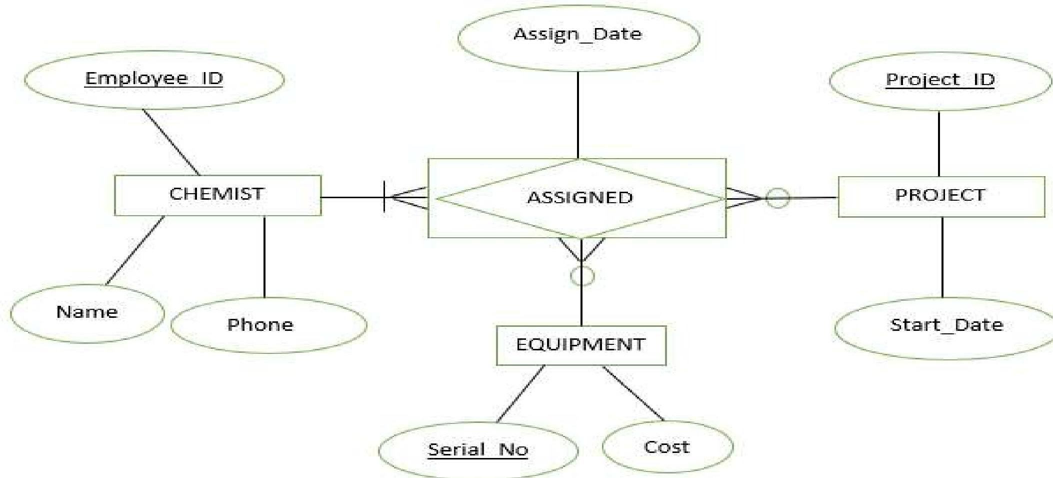


Figure: Binary Many-to-Many Relationship

Employee_ID	Name	Phone

Project_ID	Start_Date

Employee_ID	Project_ID	Serial_No	Assign_Date

Serial_No	Cost

Figure: Relations in Ternary Relationship

Week No. 10 & 11: Concurrency, Recovery and Integrity

Concurrency:
Concurrency is a **situation** in which two **or** more users **access** the **same piece** of **data** at the **same time**. In a **multi-user environment**, **concurrency** occurs very **commonly**. In some **situations**, the **concurrent** access may **arise** to some **serious problems**. The **ability** of a **database system** which handles simultaneously **or** a number of transactions by **interleaving** (inserting/adding) **parts** of the **actions** or the **overlapping**, this is called **Concurrency** of the **system**.

Concurrency Control:
Concurrency Control is **important** because the **simultaneous** execution of **transactions** over a **shared database** can create several **data integrity** and **consistency** problems. **Concurrency Control** is the **process** of **managing** simultaneous operations on the **database** without having them **interfere** with **one another**.

A **major objective** in developing a **database** is to enable many users to access **shared data concurrently**. **Concurrent access** is relatively **easy** if **all users** are only **reading data**, as there is **no way** that **they** can **interfere** with **one another**.

However, when two **or** more **users** are accessing the **database simultaneously** and at least **one** is **updating data**, there may be **interference** that can **result** in **inconsistencies**. Although **two transactions** may be **perfectly correct** in **themselves**, the **interleaving** (inserting something between two things) of **operations** sometimes may **produce** an **incorrect result**, thus can create **problem** in the **integrity** and **consistency** of the **database**.

Concurrency Problems: Different **problems** that may **occur** due to **Concurrency** are as **follows**:

1. Lost Updates/Lost Update Problem
2. Uncommitted Data/ Uncommitted Dependency (or dirty read) Problem
3. Inconsistent Retrievals/ Inconsistent Analysis Problem.

1. Lost Update Problem:
The **problem** arises when two **or** more **transactions** update the same data **concurrently**. It occurs when two **or** more **transactions** select the **same row** and then **update** the **row** based on the **value originally selected**. Each **transaction** is **unaware** of **other transactions**. The **last update** overwrites **updates** made by the **previous transactions**. It **results** in **loss** of **data**.

Example No. 01: Suppose there is a **stock table** that contains three attributes: **Product Code**, **Description** and **Quantity**. There are **two teams** of staff, **Team A** and **Team B**. **Team A** is **responsible** for **stock-in products** and **Team B** is **responsible** for **stock-out products**.

Table: Lost Update Problem

Time	Team A	Stock Table	Team B
9:00		Qty = 100	
10:30	Retrieve Qty. Qty = 100.		Retrieve Qty. Qty = 100.
10:31			Update Qty = Qty - 90
10:32		Qty = 10	
10:33	Update Qty = Qty + 30		
10:34		Qty = 130	

Result: **Team B's update** is **lost** at **10:33** which is **overwritten** by **Team A**'s **update**.

Example No. 02:
The **Lost Update Problem** occurs when **two concurrent** transactions, **T1** and **T2**, are **updating** the **same data element** and **one** of the **updates** is **lost** (overwritten by the other transaction).

Two concurrent transactions update PROD_QOH:

Transaction	**Operation**
T1: Purchase 100 units	PROD_QOH = PROD_QOH + 100
T2: Sell 30 units	PROD_QOH = PROD_QOH – 30

Transaction T1 is executing **concurrently** with **Transaction T2**. **T1** is withdrawing **£10** from an **account** with balance **balx**, initially **£100**, and **T2** is depositing **£100** into the **same account**. Transactions **T1** and **T2** start at nearly the **same time**, and **both** read the **balance** as **£100**. **T2** increases **balx** by **£100** to **£200** and stores the **update** in the **database**. **Meanwhile**, transaction **T1** decrements its copy of **balx** by **£10** to **£90** and stores **this value** in the **database**, overwriting the **previous update**, and thereby **'losing'** the **£100** previously **added** to the **balance**.

Result: T2 update is **lost** which is **overwritten** by **T1 update**.

2. Uncommitted Data/ Uncommitted Dependency (or Dirty Read) Problem:
This **problem** arises when two **or** more **transactions** work on the **same table**. **One transaction** retrieves **or** updates **certain part** of **data** before **other transaction** rollbacks update on the **same data**. The **uncommitted dependency** occurs when a **second transaction** selects a **row** that is being updated by **another transaction**. The **second transaction** is reading data that has **not** been **committed** yet and may be changed by the **transaction** that is updating the **row**.

Example No. 01:
Suppose the **stock table** contains a field **Quantity** with **value 100**. He following **sequence** of **actions** may result in **wrong result**.

Table: uncommitted Dependency Problem

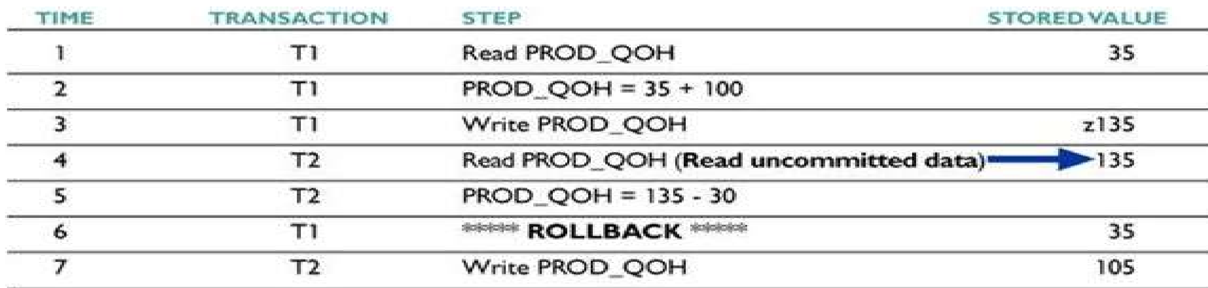
Time	Team A	Stock Table	Team B
t1		Qty = 100	
t2			Update Qty to 150
t3	Retrieve Qty	Qty = 150	
t4		Qty = 100	Rollback

In the **above example**, **Team B** updates the **value** of **Qty** to **150** at **t2**. **Team A** then retrieves this updated **value** at **t3**. **Team B** rollbacks the **action** while making **Qty** to **100** again. **So**, the **value** of **Qty** retrieved by **Team A** becomes **wrong** at **t4**.

Example No. 02:
The **phenomenon** of **uncommitted data** occurs when **two transactions**, **T1** and **T2**, are executed **concurrently** and the **first transaction** (**T1**) is **rolled back** after the **second transaction** (**T2**) has already accessed the **uncommitted data**, thus **violating** the **isolation property** of **transactions**.

Transaction	**Operation**
T1: Purchase 100 units	PROD_QOH = PROD_QOH + 100 (Rolled back)
T2: Sell 30 units	PROD_QOH = PROD_QOH – 30

Example No. 03:
The **following Table** shows **how** the **Uncommitted Data Problem** can arise when the **ROLLBACK** is completed after **T2** has begun its execution.



TIME	TRANSACTION	STEP	STORED VALUE
1	T1	Read PROD_QOH	35
2	T1	PROD_QOH = 35 + 100	
3	T1	Write PROD_QOH	z135
4	T2	Read PROD_QOH (**Read uncommitted data**) →	135
5	T2	PROD_QOH = 135 - 30	
6	T1	***** **ROLLBACK** *****	35
7	T2	Write PROD_QOH	105

The **Uncommitted Dependency Problem** occurs when **one transaction** is allowed to see the **intermediate results** of **another transaction** before it has **committed** (performed).

Example No. 04:
Uncommitted Dependency can **cause** an **error**, using the **same initial value** for balance **balx** as in the **previous example**. Here, **transaction T4** updates **balx** to **£200**, but it **aborts** the **transaction** so that **balx** should be restored to its **original value** of **£100**. **However**, by this time **transaction T3** has read the **new value** of **balx** (**£200**) and is using **this value** as the **basis** of the **£10** reduction, giving a **new incorrect balance** of **£190**, instead of **£90**. The **value** of **balx** read by **T3** is called **Dirty Data**, giving rise to the **alternative name**, the **Dirty Read Problem**.

3. Inconsistent Retrievals/ Inconsistent Analysis Problem:

Inconsistent Retrievals occur when a **transaction** accesses **data** before and after **another transaction**(s) finish working with **such data**. For **example**, an **Inconsistent Retrieval** would occur if **transaction T1** calculated total function over a **set** of **data** while another **transaction** (**T2**) was updating the **same data**. The **problem** is that the **transaction** might **read** some data before they are changed and **other data** after they are changed, thereby yielding **inconsistent results**. The **problem** of **inconsistent analysis** occurs when a **transaction** reads **several values** from the **database** but a **second** transaction updates some of them during the execution of the first.

Example No. 01:

Three Accounts:	Acc-01: 40	Acc-02: 50	Acc-03: 30	Sum = 0

Time	T1	T2
t1	Retrieve Acc-1 Sum = Sum+Acc-01 = 40	-
t2	Retrieve Acc-2 Sum = Sum+Acc-02 = 90	-
t3	-	Retrieve Acc-3
t4	-	Update Acc-3 = 0
t5	-	Retrieve Acc-1
t6	-	Update Acc-1 = 50
t7	-	Commit
t8	Retrieve Acc-3 Sum = Sum+Acc-03 = 90	-
t9	Commit	-

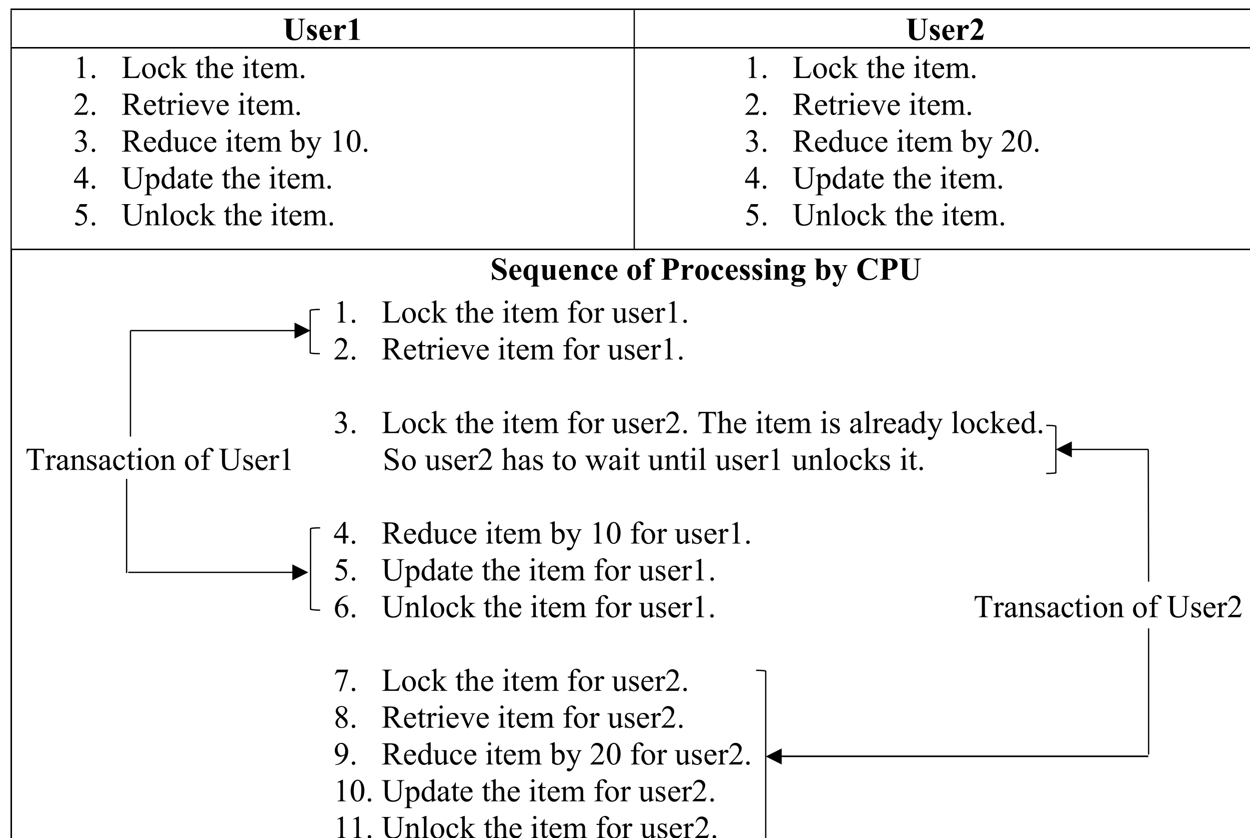
At the time **t9**, the **value** of **sum** is **90**, which is **wrong**.

Example No. 02:
For **example**, a **transaction** that is **summarizing data** in a **database** (for example, totaling balances) will obtain **inaccurate results** if, while it is executing, **other transactions** are updating the **database**. One **example** can be, in which a **summary** transaction **T6** is executing **concurrently** with transaction **T5**. Transaction **T6** is **totaling** the **balances** of **account x** (£100), **account y** (£50), and **account z** (£25). **However**, in the **meantime**, transaction **T5** has transferred **£10** from **balx** to **balz**, so that **T6** now has the **wrong result**.

This **problem** is avoided by **preventing** transaction **T6** from reading **balx** and **balz** until after **T5** has completed its updates.

Concurrent Solutions:
The three most common problems in concurrent transaction execution are lost updates, uncommitted data, and inconsistent retrievals. **Concurrency controls** can be used to avoid those problems as by utilizing **locking methods**, they facilitate the isolation of data items used concurrently executed transactions, as **locks** guarantee exclusive use of a data item to a concurrent transaction.

Resource Locking:
One way to prevent **concurrency problems** is to **lock** the **shard data**. **Locking** ensures that the **shared data** can be used by **one user** at **one time**. When a **user** accesses the **data**, the **second user** has to **wait** until the **first user** finishes **his work**. **Suppose** there is an **item** in the **database** with a **value 100**. If **two users** try to **access** the **item** to **update** it, the **locking mechanism** will **work** as **follows**:



User1	User2
1. Lock the item. 2. Retrieve item. 3. Reduce item by 10. 4. Update the item. 5. Unlock the item.	1. Lock the item. 2. Retrieve item. 3. Reduce item by 20. 4. Update the item. 5. Unlock the item.

Figure: Locking Mechanism
After both transactions are completed, the value of item will be **70**. The locking mechanism ensures that if one transaction is in progress, the second transaction has to wait for the completion of first.

Schedules:
The **Schedule** is **responsible** for **Concurrency Control**. The **Schedule** would try to prevent **conflicts** by doing the **First Come First Serve** (FCFS) in the **database**. The **second way** the **schedule** used to resolve **conflict** is by facilitating **data isolation** to make sure that the **transactions** do not **update** the **same** data element.

Two types of Schedules:

i. Serial Schedule:
Serial Execution is an **execution** where **transactions** are executed in a **Sequential Order**, that is, one after another. A **transaction** may **consist** of **many operations**. **Serial Execution** means that **all** the **operations** of **one transaction** are executed **first**, followed by **all** the **operations** of the **next transaction** and like that. A **Schedule** or **History** is a **list** of **operations** from one **or** more **transactions**. A **Schedule** represents the **order** of **execution** of **operations**. The **order** of **operations** in a **Schedule** should be the **same** as in the **transaction**. **Schedule** for a **serial execution** is called a **Serial Schedule**, so in a **Serial Schedule, all operations** of **one transaction** are **listed** followed by all the operations of **other transactions** and **so on**. With a given **set** of **transactions**, there could be different **Serial Schedules**. For **example**, if there are **two transactions**, then there could be two different **Serial Schedules** as is **explained** in the **table** below:

Serial Schedule 1 TA, TB	**BAL**	**Serial Schedule 2 TB, TA**	**BAL**
Read $(BAL)_A$	50	Read $(BAL)_B$	50
$(BAL = BAL - 10)_A$	40	$(BAL=BAL * 2)_B$	100
Write $(BAL)_A$	40	Write $(BAL)_B$	100
Read $(BAL)_B$	40	Read $(BAL)_A$	100
$(BAL=BAL * 2)_B$	80	$(BAL = BAL - 10)_A$	90
Write $(BAL)_B$	80	Write $(BAL)_A$	90

Table: Two different Serial Schedules, TA, TB and TB, TA

The **table** shows **two different schedules** of **two transactions** TA and TB. **Serial Schedule** is a plan to execute transactions **serially**. The **internal sequencing** of each transaction is preserved (well maintained). A **Serial Schedule** ensures that **each transaction** executes as if it is the only one accessing the **database** at **one time**.

ii. Serializable Schedule:
Serializable Schedule is a **non-serial schedule** in which **transaction operations** are **interleaved** while ensuring **consistency**. An **interleaved schedule** is said to be **serializable** if its **final state** is **equivalent** to some **Serial Schedule**. **Serializable Schedule** can also be **defined** as a **schedule** in which **conflicting operations** are in a **Serial Order**. The **Serializable Schedule** ensures the **consistency** of the **database**. **However**, this is worthwhile to mention here again that **serializability** takes **care** of the **inconsistencies** only **due** to the **interleaving** of **transactions**. The **serializability** concerns generating **Serializable Schedules**. It has no **read** and **write** conflicts which can affect **database consistency**. It also does not **suffer** from **issues** such as **lost update**, **dirty read** etc.

Database Failure:
In every **DBMS**, there is a possibility of **hardware** or **software failure**. The **failures** may occur without **warning**. The **data** in the **database** may be **lost** or **damaged** due to the **failure** of **DBMS**. After a **failure** occurs, a **DBMS** should **recover** the **information** that was entered into the **database**. The most **common reasons** of **Failure** are: Failure of computer system, Breakdown of hardware, Program bugs, User mistakes.

Database Failure Classification/Types:

Generally, there are **three types** of **Failures** as **follows**:

1. Transaction Failure
2. System Failure
3. Media Failure/Disk Failure

1. Transaction Failure:
A **transaction** has to **abort** (terminate) when it **fails** to execute or when it reaches a point from where it can't go any further. This is called **Transaction Failure** where only a few transactions or processes are hurt. **Transactions** may **fail** because of incorrect input, deadlock, incorrect synchronization.

Reasons for a Transaction Failure could be:

- **Logical Errors**:

Where a **transaction** cannot complete because it has some **code error** or any **internal error**.

- **System Errors:**

Where the **database system** itself terminates an **active transaction** because the **DBMS** is not able to execute it, or it has to stop because of some **system condition** (disorder). For **example**, in case of **deadlock** or **resource** unavailability, the **system** aborts an **active transaction**.

2. System Failure:
A **System Failure** is also known as **Instance Failure**. It is a **failure** of the **main memory** of a **computer system**. **System Failures** may be caused by a **power failure**, an **application** or **operating system** crash, **memory error** or some other reason. The **end result** is the unexpected termination of the **Database Management System** (DBMS) software.

3. Media Failure/Disk Failure:
A **Media Failure** is also known as **Instance Failure**. It is a **failure** of the **disk storage system** of a **computer system**. In early days of technology evolution, it was a common problem where **hard-disk drives** or **storage drives** used to **fail** frequently. **Disk Failures** include **formation** (creation) of **bad sectors**, **unreachability** to the **disk**, disk head crash or any other failure, which destroys all or a part of **disk storage**.

Database Recovery and Recovery Techniques:
DBMS is a highly complex system with hundreds of transactions being executed every second. If it fails or crashes amid (among) transactions, it is expected that the system would follow some sort of algorithm or techniques to recover the lost data.

Important Terms:

Atomicity: A **transaction** must be an **atomic** unit of work. A transaction must completely succeed or completely fail. If any statement in transaction fails, the entire transaction fails completely.
Consistency: A **transaction** must **leave** the **data** in a **consistent state** after completion.
Isolation: All **transactions** that modify the **data** are **isolated** from each other. They don't access the **same data** at the **same time**.
Durability: The **durability** means that the **modifications** made by a **transaction** are permanent and persistent (determined). If the **system** is crashed or rebooted, **data** should be guaranteed to be completed when the computer restarts.
Old values: before image (BFIM)
New values: after image (AFIM)
Undo: Restore all BFIMs on to disk (Remove all AFIMs).
Redo: Restore all AFIMs on to disk.
Write ahead logging: BFIM of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before the BFIM is overwritten with the AFIM in the database on disk.
Force writing: If all data updated by a transaction are immediately written to disk when the transaction commits, it is called a force writing.
Committed transactions: Transactions that have completed before the time of failure.
Active transactions: That have started but not committed at the time of failure.
Immediate Update: As soon as a data item is modified in cache, the disk copy is updated.
Deferred Update: All modified data items in the cache is written either after a transaction ends its execution or after a fixed number of transactions have completed their execution.

System Log/Transaction Log:
The **system** must keep **information** about the **changes** that were applied to **data items** by the **various transactions**. This **information** is typically kept in the **System Log**. **Log** is a **sequence** of **records**, which maintains the **records** of **actions** performed by a **transaction**. It is important that the **logs** are written **prior** (previous) to the **actual modification** and stored on a stable storage media, which is failsafe.
Log entries are **sequential** in **nature**. The **Transaction Log** is split up into **small chunks** (portions) called **Virtual Log Files**. When a **Virtual Log File** is **full**, **transactions** automatically move to the **next Virtual Log File**.

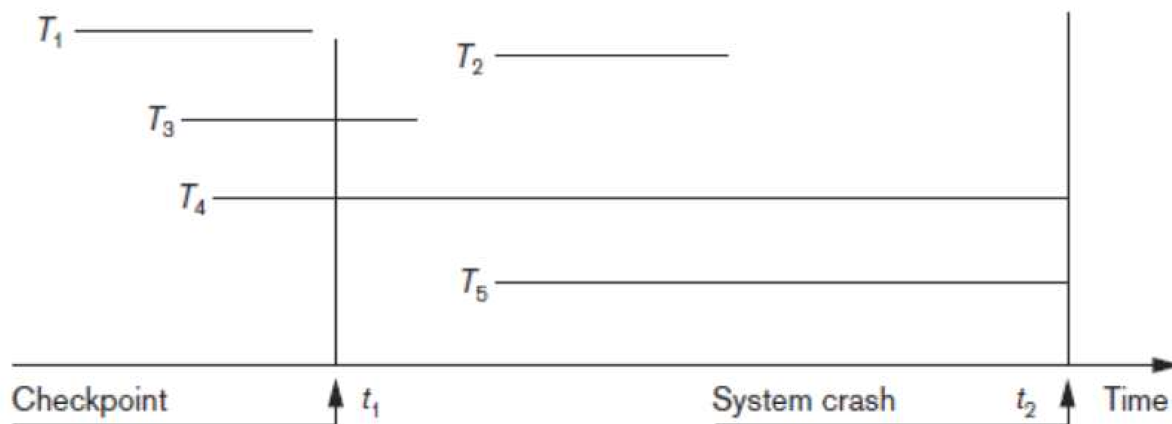
Log Buffer: Stored at volatile memory (main memory).
Log File: Stored at non-volatile memory (disk).

Types of entries in System Log:

1. **[start_transaction, T]:** Indicates that transaction **T** has started execution.
2. **[write_item, T, X, old_value, new_value]:** Indicates that transaction **T** has changed the **value** of **database** item **X** from **old_value** to **new_value**.
3. **[read_item, T, X]:** Indicates that transaction T has read the value of database item X.
4. **[commit, T]:** Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. **[abort, T]:** Indicates that transaction T has been aborted.

Checkpoint:

A [**checkpoint**] record is written into the **log** periodically. At this **point**, system writes out to the **database** on disk all DBMS buffers that have been modified. All **transactions** having [commit, T] entries in the **log** before a [**checkpoint**] entry need not to be redone in case of a system crash. **Checkpoints** are used in conjunction (combination) with **transaction logs**. **Checkpoint** is a **mechanism** where all the **previous logs** are removed from the **system** and stored permanently in a storage disk. **Checkpoint** declares a **point** before which the **DBMS** was in consistent state, and all the transactions were committed. A **checkpoint** is a **marker** that indicates the **last time** a **database** and **transaction log** were **synchronized**. If the **database** must be **restored**, only after – images for transactions that began after the **last checkpoint** need to be applied.



System restart will restore **Trans 1** successfully that is committed before **checkpoint**. The **Tran 2** will need to be **redone** using **forward** (rollforward) **recovery** as it started after the **checkpoint** and was **committed** before the **crash**. **Trans 3** also will need to be **redone** using **forward** (rollforward) **recovery** as it was started before the **checkpoint** but **committed** before the **crash**. So, both **Trans 2** and **Trans 3** are part of a **redo list**. **Trans 4** and **Trans 5** will need to be **undone** using **backward** (rollback) **recovery** as **both** were **uncommitted** at the time of the **crash**.

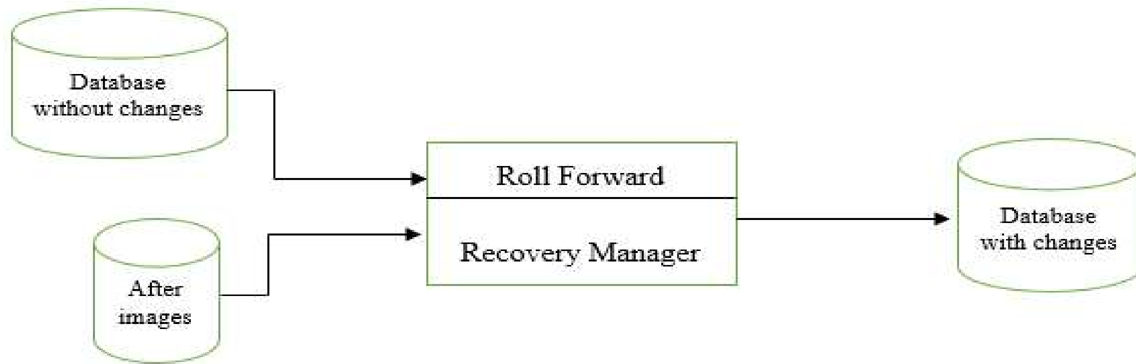
Database Recovery Techniques:
When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature.

There are three types of Recovery Techniques:

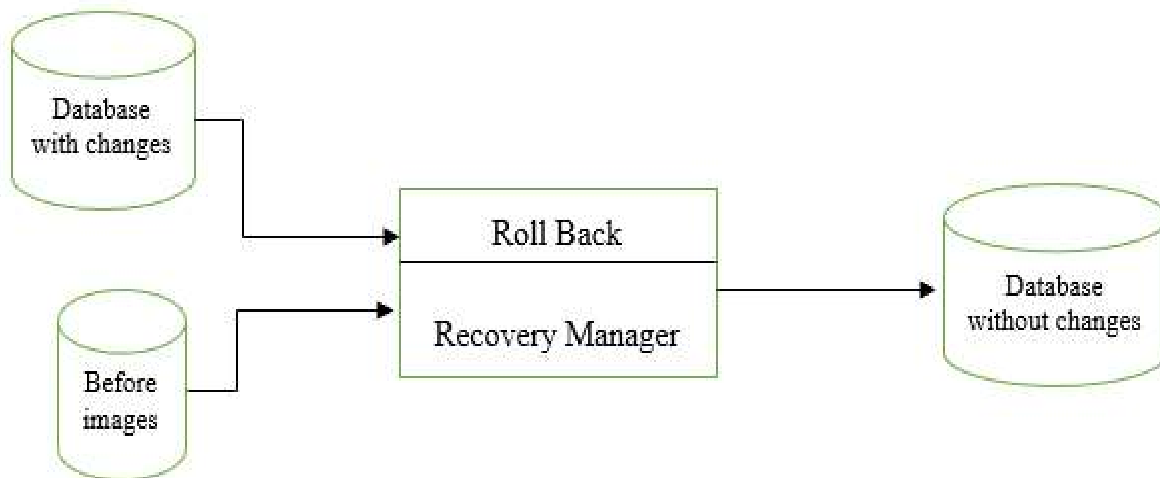
- Recovery by Transaction Rollback (Undo) and Rollforward (Redo)
- Recovery Based on Deferred Update
- Recovery Techniques Based on Immediate Update

Recovery by Transaction Rollback (Undo) and Rollforward (Redo):
In this strategy, a **transaction log file** is created. All **transactions** that make changes to the **database**, are **recorded** in **transaction log** before they are applied to the **database**. If there is any **failure** in the **system**, the **log file** is used to **reapply** all changes to the **database**.

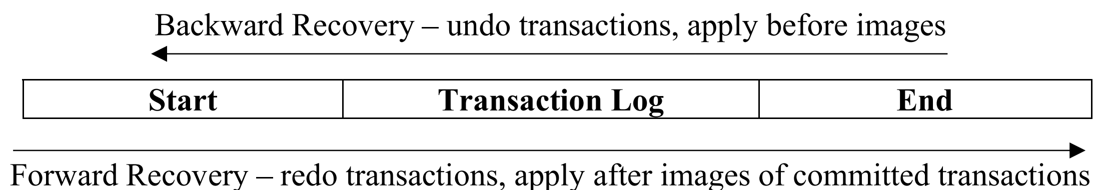
A **database** that has been completely **corrupted**, may require **forward recovery** from the **last backup**. This **process** would apply **after images** of **committed transactions** to the **backup** copy.



Secondly, the **method** of **rollback** is **applied**. In this **process**, **all changes** made by **different transactions** are **undone**. Then the **valid transactions** that were in **process** at the **time** of **failure** are **restored**. **Transactions** that were **incomplete** at the **time** of a **failure** are **identified** and the **before image** is applied to the **database**.



Both **rollforward** and **rollback** require **log file**. All **changes** are stored in **log file** before the **changes** are applied to the **database**.



Some **Recovery Manager** (RM) use **strategies** which involve the **building** of **undo** and **redo lists**. The **redo list** may only contain the **most recent committed transactions** for a **given record**. The **undo list** may only contain the **earliest transaction** for a **given record**.

Example: The read and write operations of three transactions:

T_1	T_2	T_3
read_item(A)	read_item(B)	read_item(C)
read_item(D)	write_item(B)	write_item(B)
write_item(D)	read_item(D)	read_item(A)
	write_item(D)	write_item(A)

System log at point of crash:

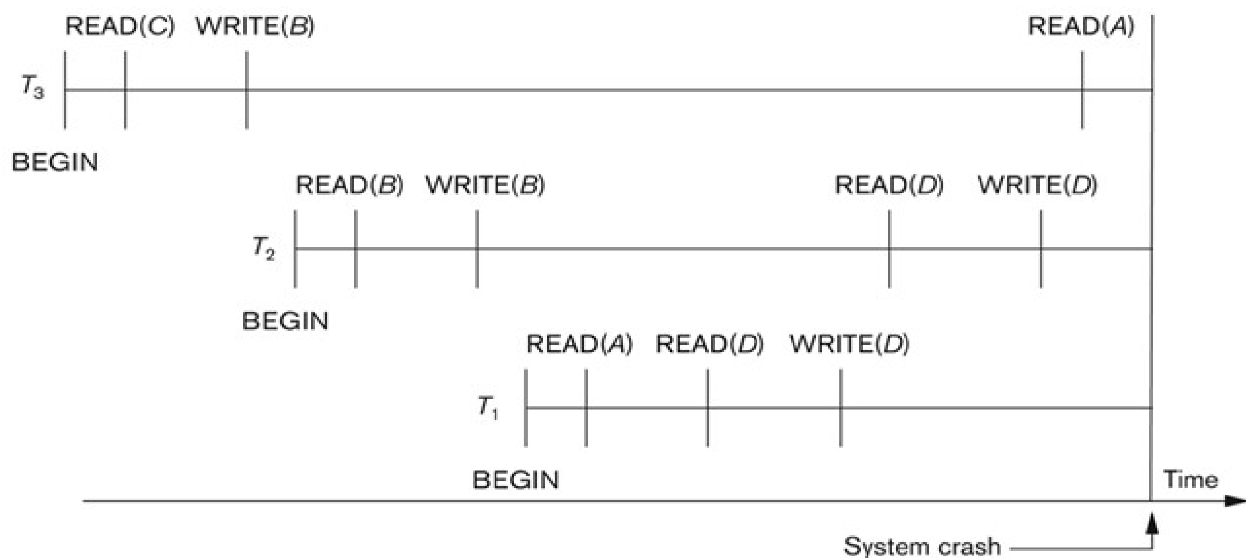
		A	B	C	D
		30	15	40	20
	[start_transaction,T_3]				
	[read_item,T_3,C]				
*	[write_item,T_3,B,15,12]		12		
	[start_transaction,T_2]				
	[read_item,T_2,B]				
**	[write_item,T_2,B,12,18]		18		
	[start_transaction,T_1]				
	[read_item,T_1,A]				
	[read_item,T_1,D]				
	[write_item,T_1,D,20,25]				25
	[read_item,T_2,D]				
**	[write_item,T_2,D,25,26]				26
	[read_item,T_3,A]				

← System crash

* T_3 is rolled back because it did not reach its commit point.

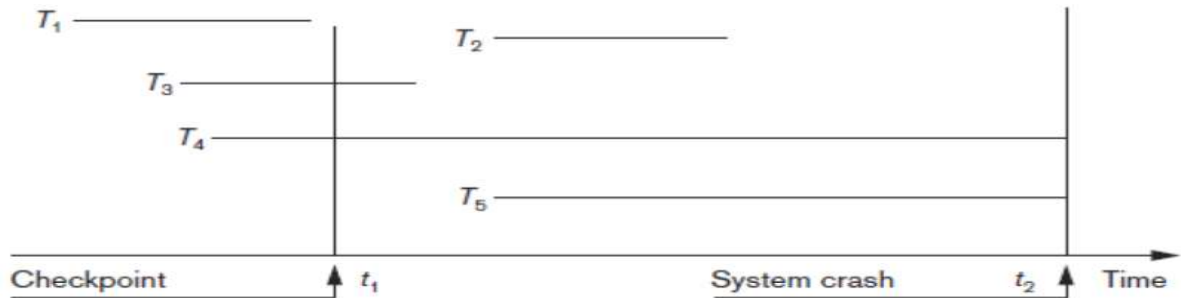
** T_2 is rolled back because it reads the value of item B written by T_3.

Operations before the crash:



Recovery Based on Deferred Update:

Also **called No-UNDO/REDO** approach. The **idea** is to **postpone** any **actual updates** to the **database** on **disk** until the **transaction** reaches its **commit point**. During **transaction execution**, the **updates** are **recorded** only in the **log** and **force** written to **disk** only after **transaction** reaches its **commit point**.
If a **transaction** fails before reaching its **commit point**, there is **no** need to **undo** any **operations** because the **transaction** has not **affected** the **database** on **disk** in any way. It means that after **reboot** from a **failure** the **log** is used to **redo** all the **transactions** affected by this **failure**. **No undo** is required because **no AFIM** is flushed to the **disk** before a **transaction commits**.

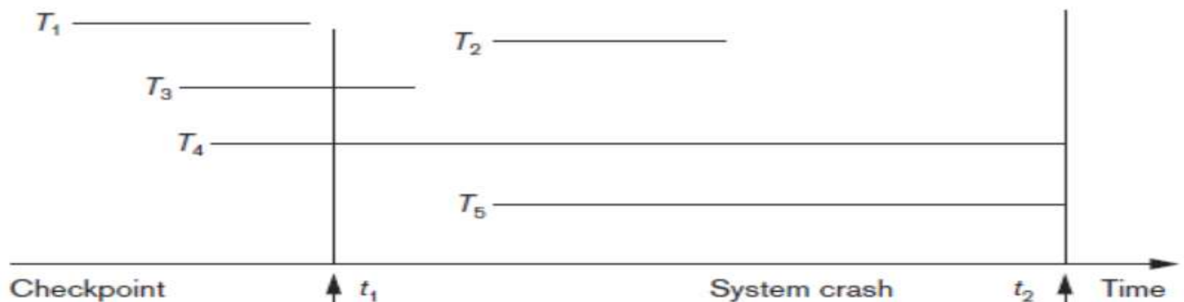


System restart will restore **Trans 1** successfully that is **committed** before **checkpoint**. The **Tran 2** will need to be **redone** using **forward** (rollforward) **recovery** as it started after the **checkpoint** and was **committed** before the **crash**. **Trans 3** also will need to be **redone** using **forward** (rollforward) **recovery** as it was started before the **checkpoint** but **committed** before the **crash**. So, **both Trans 2** and **Trans 3** are part of a **redo list**. **Trans 4** and **Trans 5** will no need to be **undone** because **no AFIM** is flushed to the **disk** before a **transaction commits**.

Recovery Techniques Based on Immediate Update:
when a **transaction** issues an **update command**, the **database** on **disk** can be updated **immediately**, without **waiting** for the **transaction** to **commit**. It is not **mandatory** that every **update** applied **immediately** to **disk**. It is possible that **some updates** are applied to **disk** before the **transaction commits**. In this **algorithm**, **AFIMs** of a **transaction** are flushed to the **database** disk before it **commits**. For this **reason**, the **Recovery Manager** undoes **all transactions** during **recovery**. No **transaction** is **redone**.

Two Variations:

- **UNDO/NO-REDO:** If the **recovery technique** ensures that all updates of a transaction are recorded in the database on disk before the transaction commits, **REDOing** is not required.
- **UNDO/REDO:** If the **transaction** is **allowed** to **commit** before all its changes are written to the database. **Example:**



System restart will restore **Trans 1** successfully that is **committed** before **checkpoint**. The **Tran 2** will need to be **redone** using **forward** (rollforward) **recovery** as it started after the **checkpoint** and was **committed** before the **crash**. **Trans 3** also will need to be **redone** using **forward** (rollforward) **recovery** as it was started **before** the **checkpoint** but **committed** before the **crash**. So, both **Trans 2** and **Trans 3** are part of a **redo list**. **Trans 4** and **Trans 5** will need to be **undone** using **backward** (rollback) **recovery** as **both** were **uncommitted** at the **time** of the **crash**.

Integrity Control System:
Integrity Controls are **mechanisms** and **procedures** that are **built** into a **system** to **safeguard** the **system** and the **information** within it. Some of the controls—called **Integrity Controls**—must be **integrated** into the **Application Programs** that are being developed and the **database** that supports them. **Integrity controls** ensure **correct system function** by **rejecting** invalid data inputs, preventing unauthorized data outputs, and protecting data and programs against accidental or malicious tampering (interfering).

The primary objectives of Integrity Controls System are to:

- Ensure that only appropriate and correct business transactions occur.
- Ensure that the transactions are recorded and processed correctly.
- Protect and safeguard the assets of the organization (including hardware, software, and information).

Constraints:
Constraints are **restriction** on **database** which ensure that the **data** is accurate.

Types of Integrity Constraints:

i. Not Null (Required Data)
ii. Entity Integrity (Primary Key Constraint)
iii. Referential Integrity (Foreign Key Constraint)
iv. Domain Constraints
v. General Constraints

i. Not Null Constraint:

- **Null:**

Represents a **value** for an **attribute** that is currently **unknown** or is not applicable for this tuple. However, a **null** is not the **same** as a zero numeric value or a text string filled with spaces, zeros and spaces are values, but a **null** represents the **absence** of a **value**.

- **Not Null Constraint:**

The **value** of an **attribute** cannot be **null** during the **insertion** of a **record**.

ii. Entity Integrity:
This is related to the concept of **Primary Keys**. **All tables** should have their own **Primary Keys** which should **uniquely** identify a **row** and **not** be **NULL**. **Entity Integrity** means, in a (base) **relation**, **no attribute** of a **Primary Key** can be **null**.
By **definition**, a **Primary Key** is a **minimal identifier** that is used to **identify** tuples **uniquely**. This means that **no subset** of the **Primary Key** is **sufficient** to provide **unique identification** of **tuples**. If a **null** allows for any **part** of a **Primary Key**, we are implying (suggesting) that not all the attributes are needed to **distinguish** between **tuples**, which contradicts (denies) the **definition** of the **Primary Key**.

iii. Referential Integrity:

This is related to the concept of **Foreign Keys**. A **Foreign Key** is a **key** of a **relation** that is **referred** in another **relation**. If a **Foreign Key** exists in a **relation**, either the **Foreign Key** value must **match** a **Candidate Key** value of **some tuple** in its **home relation** or the **Foreign Key** value must be **wholly null**.

EmployeeID	Name	HouseNo_alloted
1001	Ali	7
1002	Asghar	Null
1003	Akbar	11

HouseNo	Address
6	Coffee shop
7	House no 7, near Usmania Mosque
11	House no 11, near Usmania Mosque

iv. Domain Integrity - This means that there should be a defined **domain** for all the **columns** in a **database**. **Restricting** an **attribute** to its **domain values**.

A **domain** is defined as the **set** of **all unique values** permitted for an **attribute**. For **example**, a **domain** of **date** is the **set** of all possible **valid dates**, a **domain** of **integer** is all possible **whole numbers**, a **domain** of **day-of-week** is **Monday** to **Sunday** (7 days).

v. General Constraints:

Additional rules specified by the **users** or **Database Administrators** of a **database** that define or constrain some **aspect** of the **enterprise**.

Database Security:

Database Security refers to the process of protects and safeguards the database from unauthorized access or cyber-attacks. There are different types of **Database Security** such as encryption, authentication, backup, application security and physical security which should implement in your business.

Types of Database Security:

The main purpose of **Database Security** is to keep secure sensitive information of a database and maintain the database confidentiality, integrity, and availability. The types of **Database Security** are **key techniques** which are used to provide the **Database Security**.

Database Security is important to protect from cyber-attacks which can lead to financial loss, damage of brand reputation, business continuity and customer confidence.

The main Security types of a database are as follows:

1. Authentication
2. Authorization
3. Database Encryption
4. Backup Database
5. Physical Security
6. Application Security
7. Access Control
8. Web Application Firewall
9. Use Strong Password
10. Database Auditing

1. Authentication

Database Authentication is the type of **Database Security** that verify the user's login credentials which stores in database. If user's login credentials match in database, then user can access the database. That means the user has authentication to login into your database.

Authentication can be done at the operating system level or even the database level itself. **Digital Signatures** are used to verify the authenticity of data i.e., Password based Authentication.

Many other **Authentication** systems such as retina scanners or bio-metrics are used to make sure unauthorized people cannot access the database.

If an authentic user has some privilege to access the data, then he can't access the other data which are out of privilege. No unauthorized or malicious user can't login into your database. So, database authentication plays an important role for ensure **Database Security**.

2. Authorization:

Authorization is a **privilege** provided by the **Database Administer**. Users of the database can only view the contents they are authorized to view.

The different permissions for Authorizations available are:

Primary Permission:	This is granted to users publicly and directly. Grants the authorization ID directly.
Secondary Permission:	This is granted to groups and automatically awarded to a user if he is a member of the group.
Public Permission:	This is publicly granted to all the users.
Context Sensitive Permission:	This is related to sensitive content and only granted to a select users. Grants to the trusted context role.

3. Database Encryption

Encryption is one of the most effective **types** of **Database Security** which protect your database from unauthorized access during storing and transmission over the **internet**.

There are different types of **encryption algorithm** such as **AES, MD5, and SHA 1** which are used to **encrypt** and **decrypt** the all types of sensitive data.

Typically, an **encryption algorithm** transforms the plain text data into ciphertext of unreadable formats within a database. So, if hackers get access your database, then they can't use your data until the data is decrypt.

It is highly recommended to you that encrypt your sensitive data while storing into database because it provides security and protect from cyber-attacks.

4. Backup Database

Backup is another type of **Database Security** which used to restore data in case of data loss, data corruption, hacking, or natural disasters. It copying or archiving the database in real time on a secondary storage.

If you configured the **Primary** and **Secondary Servers** at same place and if the **Primary Server** is destroyed then there has a chance to destroy the **Secondary Server**. So, you can't run your application and your system will shut down until you recover.

That's why it is suggested that, always configure the **Secondary Server** physically in separate location in order to ensure **Database Security**. In that case, if the **Primary Server** is down then you can recover database from **Secondary Server**.

There are different **types** of **database backup** such as full backup, differential and incremental backup.

5. Physical Security

Physical Database Security is the protection of database server room in order to protect from unauthorized access. **Database Server** should be located in secured and climate-controlled environment in a building.

Only **DBA** (Database Administration) and **Authorized IT** (Information Technology) **Officer** can enter into the **Server Room**. If your **Database Server** is in **cloud data center** then your service provider will take necessary action to secure your database. In that case, before hosting your database in a cloud you can ask them how they will secure your database?

It is also suggested that, if possible then don't host the database server and application on the same server. Both servers should physically isolated for security purposes and performance also. Even you can make a policy for database server room which may include room is locked all times, only authorized IT Officer can check the server room environment etc.

6. Application Security

Application and **Database** have to secure in order to protect from web attacks such as SQL injection. **SQL injection** is the most common web attacks where hacker control application's database to hack sensitive information or destroy the database.

In this **technique**, the **attacker** adds the malicious code in SQL query, via web page input. It is occurring when an application fails to properly sanitize the SQL statements. So, **attacker** can add their own malicious SQL statements to access your database for malicious purposes.

To protect from **SQL injection attacks**, you can secure application by applying the following prevention methods:

- Use of Prepared Statements
- Use a Web Application Firewall
- Updating system
- Validating user input
- Limiting Privileges

7. **Access Control**

To ensure of **Database Security** you have to restrict the access of database from unauthorized users. Only **Authorized User** can get access the database and **unauthorized user** can't access the database. **Create** user accounts by **DBA** who will access the database and **set** a role and limit what they can access in your database.

So, **Access Control** is a **type** of **Database Security** which can secure your database by restricting unauthorized users' access.

8. **Web Application Firewall**

A **Web Application Firewall** or **WAF** is an application based cyber security tool which is the **Database Security** best practice. **WAF** has designed to protect applications by filtering, monitoring and blocking HTTP malicious traffic.

This **Database Security** measure controls who can access the application and prevent intruders from accessing the application via the internet. To secure your application from malicious users you should use a **Web Application Firewall** which will protect your application, database.

One of the following Web Application Firewalls can be used in a system:

- Fortinet FortiWeb
- Citrix NetScaler AppFirewall
- F5 Advanced WAF
- Radware AppWall
- Symantec WAF
- Barracuda WAF
- Imperva WAF

9. **Use Strong Password**

This is simple but very important tips for ensure **Database Security**. As a **DBA** or **IT Officer** should use **Strong Password** for **database login** and never share your password with others.

If you use easy password such as your mobile no, employee id, date of birth which is known to hackers and they will try to login using these passwords. As a result, you will lose your database control.

So, create a **Strong Password** for **database login** using combination of letters, numbers, special characters (minimum 10 characters in total) and change the password regularly. **For example: T#$jk67@89m*** can be a **Strong Password** for your **database login**.

10. **Database Auditing**

Auditing is very important **types** of **Database Security Control** which can help to detect and identify of unauthorized access to your **DBMS** (Database Management System).

Database Auditing regularly check the **log files** for who access the **database**, when they accessed, how long time stay there and what they did in database. It can easily find out if there is unauthorized access to database server. **Database Auditing** is a **type** of **database protection** which can provides overall monitoring for **Database Security** of an **organization**.

Week No. 12: Relational Algebra

Relational Algebra:
Relational Algebra is a **Procedural Query Language** that processes one or more relations to define another relation without changing original relations. The **operands** as well as the **result** are **relations**. The output of one operation can become the input of another operation to create nested expression in **Relational Algebra**. This **property** is known as **Closure**.

Basic Operations of Relational Algebra:

There are two categories of Operations in **Relational Algebra**:

1. **Unary Operations**
2. **Binary Operations**

1. Unary Operations:
The **operations** which involve only **one relation** are called **Unary Operations**. The following operations are **the Unary Operations**:

i. **Selection Operation**
ii. **Projection Operation**

i. Selection Operation:

The **Selection Operation** is a Unary Operation. The Selection Operator is Sigma **σ**. It acts like a filter on a **relation**. It returns only a certain number of tuples. It selects the tuples using a condition. The condition appears as subscript to **σ**. The **resulting relation** has the **same degree** as the **original relation**. However, the resulting relation may have fewer tuples than the original relation.

The general syntax is: $\sigma_{C}(R)$

$\sigma_{C}(R)$ returns only those tuples in R that satisfy **condition C**. A **condition C** may consist of any combination of **comparison** or **logical operators** that operate on the **attributes** of **R**.

- Comparison Operators: $=, <, >, \geq, \leq, \neq$
- Logical Operators: $\wedge, \vee, \neg$

Examples:

Assume a relation EMP has the following tuples:

Name	Office	Dept	Rank
Saleem	400	CS	Assistant
Junaid	220	Econ	Lecturer
Ghafoor	160	Econ	Assistant
Babar	420	CS	Associate
Saleem	500	Fin	Associate

Question No. 01: Select only those Employees who are in CS department.

$\sigma_{dept = \text{'CS'}}(EMP)$

Result:

Name	Office	Dept	Rank
Saleem	400	CS	Assistant
Babar	420	CS	Associate

Question No. 02: Select only those Employees with last name Saleem who are Assistant Professors.

$\sigma_{\text{Name = 'Saleem'} \wedge \text{Rank = 'Assistant'}}$ (EMP)

Result:

Name	Office	Dept	Rank
Saleem	400	CS	Assistant

Question No. 03: Select only those Employees who are Assistant Professors or in Economics department.

$\sigma_{\text{Rank = 'Assistant'} \vee \text{Dept = 'Econ'}}$ (EMP)

Result:

Name	Office	Dept	Rank
Saleem	400	CS	Assistant
Junaid	220	Econ	Lecturer
Ghafoor	160	Econ	Assistant

Question No. 04: Select only those Employees who are not int the CS Department or Lecturer.

$\sigma_{\neg\text{(Rank = 'Lecturer'} \vee \text{Dept = 'CS')}}$ (EMP)

Result:

Name	Office	Dept	Rank
Ghafoor	160	Econ	Assistant
Saleem	500	Fin	Associate

ii. Projection Operation:

Projection is also a Unary Operator. Th Projector Operator is pi π. It limits the attributes returned from the original relation. The resulting relation has the same number of tuples as the original relation. The degree of the resulting relation may be equal to or less than the original relation.

The general syntax is: $\pi_{\text{attributes}}$ R

Where attributes are the list of attributes to be displayed and R is the relation.

Examples:

Assume a relation EMP has the following tuples:

Name	Office	Dept	Rank
Saleem	400	CS	Assistant
Junaid	220	Econ	Lecturer
Ghafoor	160	Econ	Assistant
Babar	420	CS	Associate
Saleem	500	Fin	Associate

Question No. 01: Display only the names and departments of the employees.

$\pi_{\text{Name, dept}}$ EMP

Result:

Name	Dept
Saleem	CS
Junaid	Econ
Ghafoor	Econ
Babar	CS
Saleem	Fin

Combining Selection and Projection:
The Selection and Projection operators can be combined to perform both operations.

Question No. 02: Display the names of all employees working in the CS department.

π_{Name} (σ $_{\text{Dept = 'CS'}}$ (EMP))

Result:

Name
Saleem
Babar

Question No. 03: Show name and rank of those Employees who are not in CS department or Lecturer.

$\pi_{\text{Name, Rank}}$ (σ $_{\neg\text{(Rank = 'Lecturer'} \vee \text{Dept = 'CS')}}$ (EMP))

Result:

Name	Rank
Ghafoor	Assistant
Saleem	Associate

2. Binary Operations:
The operations which involve pairs of relations are called Binary Relation Operations. A Binary Operations uses two relations as input and produces a new relation as output.

The following Set Operations are the Binary Operations:

i. **Union**
ii. **Set Difference**
iii. **Intersection**
iv. **Cartesian Product**

i. Union:

The Union operation of two relations combines the tuples of both relations to produce a third relational if two relations contain identical tuples, the duplicate tuples are eliminated. The notation for the Union of two relations A and B is A UNION B.

It is donated by: U

If A is defined as A = { a, b, c } and set B is defined as B = { a, c, 1, 2 }, then C = A U B will return C = { a, b, c, 1, 2}.

The relations used in the Union operation must have same number of attributes. The corresponding attributes must also come from same domain. Such relations are also called Union compatible relations.

Union are Commutative Operations: A U B = B U A

Example: A U B

Following is an example of Union operation. Two relations A and B are combined together by using Union Operator.

Table A

X	Y	Z
1	A	10
2	B	20
3	C	30

Table B

X	Y	Z
1	A	10
4	D	40
5	E	50

A U B

X	Y	Z
1	A	10
2	B	20
3	C	30
4	D	40
5	E	50

Figure: Union Operation

ii. Set Difference:

The Difference operation works on two relations. It produces a third relation that contains the tuples that occur in the first relation but not in second. The difference operation can be performed on Union compatible relations. The order of subtraction is significant.

The Difference operator is not commutative. It means: A – B ≠ B – A

Example:

Following is an example of Difference operation. There are two relations A and B.

The result of A – B and B – A are as follows:

Table A

X	Y	Z
1	A	10
2	B	20
3	C	30

Table B

X	Y	Z
1	A	10
4	D	40
5	E	50

A – B

X	Y	Z
2	B	20
3	C	30

B – A

X	Y	Z
4	D	40
5	E	50

Figure: The Difference Operation

iii. Intersection:
The Intersection operation works on tow relations. It produces a third relation that only common tuples. Both relations must be Union compatible.

It is denoted by: ∩

Example:

Following is an example of Intersection operation. There are two relations A and B.

The result of A INTERSECTION B is as follows:

Table A

X	Y	Z
1	A	10
2	B	20
3	C	30

Table B

X	Y	Z
1	A	10
4	D	40
5	E	50

A ∩ B

X	Y	Z
1	A	10

Figure: Intersection Operation

iv. Cartesian Product:
The product works on two relations. It concatenates every tuple in one relation with every tuple in second relation. It is also called Cross Product. The Product of relation A with m tuples and relation B with n tuples is relation C with m x n tuples.

The product is denoted as: A X B.

The Product needs not to be Union compatible. It means that they can be of different degree. It is commutative and associative.

Table A

X	Y	Z
1	A	10
2	B	20
3	C	30

Table B

X	Y	Z
1	A	10
4	D	40
5	E	50

A X B

X1	Y1	Z1	X2	Y2	Z2
1	A	10	1	A	10
1	A	10	4	D	40
1	A	10	5	E	50
2	B	20	1	A	10
2	B	20	4	D	40
2	B	20	5	E	50
3	C	30	1	A	10
3	C	30	4	D	40
3	C	30	5	E	50

Figure: Cartesian Product Operation

Week No. 13: SQL using Oracle

Oracle:

Oracle Corporation produces **products** and **services** to meet **Relational Database Management System** needs. The main product is the **Oracle Server**, which enables the user to store and manage information by using **SQL** and **PL/SQL** engine. The **Oracle Server** supports **ANSI** standard **SQL** and contains extensions.

SQL:

SQL stands for **Structured Query Language**. **SQL** is a **programming language** used to communicate with the **server** to access, manipulate, and control data. **SQL** is not a **full-featured** programming language. It is simply a data sublanguage. It means that it only has **language statements** for **database definition** and **processing** (querying and updating).

SQL Commands and Syntax:

SQL is, fundamentally, a **programming language** designed for accessing, modifying and extracting information from **relational databases**. As a **programming language**, **SQL** has **commands** and a **syntax** for issuing those commands.

SQL commands are divided into several different types, including the following:

1. **Data Definition Language (DDL):** Data Definition Language commands (CREATE, ALTER, DROP, RENAME, TRUNCATE) are also called **Data Definition Commands** because they are used to **define** data **tables**.

2. **Data Manipulation Language (DML):** Data Manipulation Language commands (INSERT, UPDATE, DELETE) are used to manipulate data in existing tables by adding, changing or removing data. Unlike **DDL** commands that **define** how **data** is **stored**, **DML** commands operate in the **tables** defined with **DDL** commands.

3. **Data Query Language:** Data Query Language consists of just one command, **SELECT**, used to get specific data from **tables**. This command is sometimes grouped with the **DML** commands.

4. **Data Control Language:** Data Control Language commands (GRANT, REVOKE) are used to **grant** or **revoke** user access **privileges**.

5. **Transaction Control Language:** Transaction Control Language commands (COMMIT, ROLLBACK) are used to change the state of some data, for example, to **COMMIT** transaction changes or to **ROLLBACK** transaction changes.

SQL Syntax:

SQL Syntax, the **set** of **rules** for how **SQL statements** are written and formatted, is similar to other programming languages. **SQL syntax include the following:**

SQL statements start with a **SQL** command and end with a **semicolon** (**;**).

For example: SELECT * FROM customers;

This **SELECT statement** extracts **all** of the **contents** of a **table** called **customers**.

SQL was developed by **IBM**. It is endorsed as a national standard by American National Standards Institute (**ANSI**). The most widely implemented version of **SQL** is **ANSI SQL-92** standard.

SQL works with database programs like MS Access, DB2, Informix, MS SQL Server, Oracle etc.

Features of SQL:

The **following** are some **features** of **SQL**:

1. **SQL** is an **English**- like language. It uses words like SELECT, INSERT, DELETE etc.
2. **SQL** is a **non-procedural language**. The user specifies what to do, not how to do. **SQL** does not require to specify the access method to data.
3. **SQL commands** are not **case sensitive**.
4. **SQL statements** can be entered on **one** or **more lines**.
5. **Keywords** cannot be **split** across **lines** or **abbreviated**.
6. **SQL** processes **sets** of **records** rather than a **single record** at a time. The most **common form** of a **set** of **records** is **table**.
7. **SQL** can be used by a **range** of **users** like DBA, Application Programmer, Management Personnel and many other types of End Users.
8. **SQL** provides **commands** for a **variety** of **tasks** including:
   - Querying data.
   - Inserting, Updating, Deleting rows in a table.
   - Creating, Modifying, and deleting database objects.
   - Controlling access to the database and database objects.

Basic SQL Statements:
The basic **SQL statements** are as **follows**:

SELECT Statement with FROM Clause:

SELECT statement is used to **select** data from a **table**. It **displays** result in **tabular form**.

General Syntax:

SELECT column_name (s) FROM table_name;

The following three Relations (EMP, DEPT, SALGRADE) will be used for SQL statements:

Table: EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7876	ADANS	CLERK	7788	23-MAY-87	1100		20
7900	JANES	CLERK	7698	03-DEC-81	950		30
7902	FORD	ANALYST	7766	03-DEC-81	3000		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

Table: DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

Table: SALGRADE

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

Examples:

SELECT One Column:

Write a query that display EMPNO from EMP table.

SELECT EMPNO FROM EMP;

Result:

EMPNO
7369
7499
7521
7566
7654
7698
7782
7788
7839
7844
7876
7900
7902
7934

SELECT Multiple Columns:

Write a query that displays the columns EMPNO and ENAME from EMP table.

SELECT EMPNO, ENAME, JOB FROM EMP;

Result:

EMPNO	ENAME	JOB
7369	SMITH	CLERK
7499	ALLEN	SALESMAN
7521	WARD	SALESMAN
7566	JONES	MANAGER
7654	MARTIN	SALESMAN
7698	BLAKE	MANAGER
7782	CLARK	MANAGER
7788	SCOTT	ANALYST
7839	KING	PRESIDENT
7844	TURNER	SALESMAN
7876	ADANS	CLERK
7900	JANES	CLERK
7902	FORD	ANALYST
7934	MILLER	CLERK

SELECT All Columns:

Write a query that displays all columns from EMP table.

SELECT * FROM EMP;

The * symbol is used instead of column names to display all columns of a table.

Result:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7876	ADANS	CLERK	7788	23-MAY-87	1100		20
7900	JANES	CLERK	7698	03-DEC-81	950		30
7902	FORD	ANALYST	7766	03-DEC-81	3000		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

SELECT DISTINCT Statement:

DISTINCT keyword is used to **eliminate** duplicate rows from the **result** of a **SELECT** statement. If **DISTINCT** is not used, **all rows** are **returned** including **duplicates**.

DISTINCT One Column:
Write a query to displays all distinct DEPTNO from EMP table.

SELECT DISTINCT DEPTNO FROM EMP;

Result:

```
DEPNO
----------
  10
  20
  30
```

DISTINCT Multiple Columns:

Multiple Columns may be used with **DISTINCT**.

Write a query that displays distinct DEPTNO and JOB from EMP table.

SELECT DISTINCT DEPTNO, JOB FROM EMP;

Result:

```
DEPNO     JOB
--------------------------
  10      CLERK
  10      MANAGER
  10      PRESIDENT
  20      ANALYST
  20      CLERK
  20      MANAGER
  30      CLERK
  30      MANAGER
  30      SALESMAN
```

SELECT Statement with WHERE Clause:

WHERE Clause is used to **retrieve** data from a **table** conditionally. It can appear only after **FROM clause**.

General Syntax:

SELECT Column(s) FROM Table WHERE Condition;

Example:
Write a query that displays records of clerks from EMP table.

SELECT * FROM EMP WHERE JOB = 'CLERK';

Result:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7876	ADANS	CLERK	7788	23-MAY-87	1100		20
7900	JANES	CLERK	7698	03-DEC-81	950		30
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

Using Quotes:
SQL uses **single quotes** around **text values**. Most **database systems** also accept **double quotes**. **Numeric values** should not be **enclosed** in **quotes**.

SELECT Statement with ORDER BY Clause:
The **ORDER BY Clause** is used to **sort** the **rows**. The **process** of **arranging** data **or** records in a **sequence** is called **Sorting**. A **sort** can be **Ascending** or **Descending**.

SQL uses **ASC keyword** to specify **Ascending Sort** and **DESC keyword** for **Descending Sort**. If neither is specified, **ASC** is used as **default**. **ORDER BY** must always be the **last Clause** in **SELECT statement**. If the **records** contain **date values**, earliest **date** will appear first. If the **records** contain **character values**; it will be sorted **Alphabetically**.

Example No. 01:
Write a query that displays EMP table in alphabetical order with respect to name.

SELECT * FROM EMP ORDER BY ENAME;

Result:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7876	ADANS	CLERK	7788	23-MAY-87	1100		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7902	FORD	ANALYST	7766	03-DEC-81	3000		20
7900	JANES	CLERK	7698	03-DEC-81	950		30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7934	MILLER	CLERK	7782	23-JAN-82	1300		10
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30

Example No. 02:
Write a query that displays ENAME, JOB, and SAL columns of EMP table in descending order by SAL.

```
SELECT      ENAME, JOB, SAL
FROM        EMP
ORDER BY    SAL DESC;
```

The column after ORDER BY clause is not required to appear in SELECT clause also.

Result:

ENAME	JOB	SAL
KING	PRESIDENT	5000
SCOTT	ANALYST	3000
FORD	ANALYST	3000
JONES	MANAGER	2975
BLAKE	MANAGER	2850
CLARK	MANAGER	2450
ALLEN	SALESMAN	1600
TURNER	SALESMAN	1500
MILLER	CLERK	1300
WARD	SALESMAN	1250
MARTIN	SALESMAN	1250
ADANS	CLERK	1100
JANES	CLERK	950
SMITH	CLERK	800

ORDER BY Many Columns:
The ORDER BY clause can also be used with multiple columns.

Example No. 03:
Write a query that displays name and salary of all employees from EMP table. Result should be sorted in ascending order by DEPTNO and then in descending order by SAL.

```
SELECT      ENAME, SAL
FROM        EMP
ORDER BY    DEPTNO, SAL DESC;
```

Result:

ENAME	SAL
KING	5000
CLARK	2450
MILLER	1300
SCOTT	3000
FORD	3000
JONES	2975
ADANS	1100
SMITH	800
BLAKE	2850
ALLEN	1600
TURNER	1500
WARD	1250
MARTIN	1250
JANES	950

Week No. 14: Built in Functions

Functions:
Functions are used to manipulate data item. They accept one or more arguments and return one value. An argument is user-supplied constant, variable or column reference. The format for a function is as follows:

Function name (argument1, argument2……)

Aggregate Functions:
Aggregate functions generate summary value. They can be applied to all the rows in a table or the rows specified by WHERE clause. Aggregate functions generate a single value from each set of rows. Aggregate functions such as COUNT, AVG, SUM, MAX, MIN generate a summary value.

i. COUNT Function:
Count function is used to count the number of rows.

Syntax:
COUNT (*/Column_name/DISTINCT Column_name)

COUNT (*):
The COUNT (*) is used to count all the rows including rows containing duplicates and null values.

Example No. 01:
Write a query that displays total columns in EMP table.

```
SELECT    COUNT (*)
FROM      EMP
WHERE     DEPTNO = 20;
```

Result:

```
COUNT (*)
---------------
        5
```

The above example counts all the Employees in department 20 of EMP table including rows containing duplicate and null values.

COUNT (Column_name):
The COUNT (Column_name) is used to count the values in the column specified excluding any null values.

Example No. 02:
Write a query that displays total records in COMM column of EMP table.

```
SELECT    COUNT (COMM)
FROM      EMP;
```

Result:

```
COUNT (COMM)
---------------------
        4
```

The above example counts the values in COMM excluding null values.

COUNT (DISTINCT Column_name):
The COUNT (Distinct Column_name) is used to count all the rows excluding rows containing duplicate and null values.

Example No. 03:

```
SELECT    COUNT (DISTINCT MGR)
FROM      EMP;
```

Result:

```
COUNT (DISTINCT Mgr)
-------------------------------
                 6
```

The above example counts the value in MGR Column after eliminating duplicates and null values.

ii. AVG Function:
The AVG Function returns the average of all values of expression. AVG Function can be used with numeric column only and will automatically ignore the null values.

Syntax:
AVG (ALL/DISTINCT/Expression)

AVG (ALL):
It is the default and is applied to all values.

AVG (DISTINCT):
It indicates that AVG is performed only on each unique instance of a value.

AVG (Expression):
It is any valid expression like column name.

Example No. 04:
Write a query that calculates the average salary of all employees.

```
SELECT    AVG (SAL)
FROM      EMP;
```

Result:

```
AVG (SAL)
--------------
2073.2143
```

iii. SUM Function:
The SUM Function returns the sum of all values in an expression. It supports the use of DISTINCT to summarize only unique value in the expression. Null values are ignored. It can be used only with numeric columns.

Syntax:
SUM (ALL/DISTINCT)

Example No. 05:
Write a query that displays the sum of salaries of all clerks in EMP tanle.

```
SELECT      SUM (SAL)
FROM        EMP
WHERE       JOB = ‘CLERK’;
```

Result:

```
SUM (SAL)
---------------
   4150
```

iv. MAX Function:
The MAX Function returns the maximum value in an expression. It ignores all null values. It can be used with all datatypes.

Syntax:
MAX (ALL/DISTINCT/Expression

Example No. 06:
Write a query that finds the maximum salary earned by clerk.

```
SELECT      MAX (SAL)
FROM        EMP
WHERE       JOB = ‘CLERK’;
```

Result:

```
MAX (SAL)
---------------
         1300
```

v. MIN Function:
The MIN Function returns the minimum value in an expression. It ignores all null values. It can be used with all datatypes.

Syntax:
MIN (ALL/DISTINCT/Expression)

Example No. 07:
Write a query that finds the minimum salary earned by clerk.

```
SELECT      MIN (SAL)
FROM        EMP
WHERE       JOB = 'CLERK';
```

Result:

```
    MIN (SAL)
    --------------
            800
```

Example No. 08:
Write a query that finds minimum, maximum and average salaries of all employees.

```
SELECT      MAX (SAL), MIN (SAL), AVG (SAL)
FROM        EMP;
```

Result:

```
    MAX (SAL)       MIN (SAL)       AVG (SAL)
    ---------------  --------------  ---------------
    5000            800             2073.2143
```

Week No. 15: GROUP BY clause and Joining

GROUP BY:

The GROUP BY clause can be used to divide the rows in a table into smaller group. If aggregate function is used in SELECT statement, GROUP BY clause produces a single value per aggregate.

Syntax:

GROUP BY Column_name

Example No. 01:

Write a query that calculates the average salary for each different job.

```
SELECT      AVG (SAL)
FROM        EMP
GROUP BY    JOB;
```

Result:

```
AVG (SAL)
---------------
3000
1037.5
2758.3333
5000
1400
```

Example No. 02:
Write a query that finds the minimum, maximum salary for each job type.

```
SELECT      JOB, MAX (SAL), MIN (SAL)
FROM        EMP
GROUP BY    JOB;
```

Result:

JOB	MAX (SAL)	MIN (SAL)
ANALYST	3000	3000
CLERK	1300	800
MANAGER	2975	2450
PRESIDENT	5000	5000
SALESMAN	1600	1250

Excluding Rows in GROUP BY Clause:
Some rows may be pre-excluded in WHERE clause before dividing them into groups.

Example No. 03:
Write a query that displays the average salary for each Job excluding managers.

```
SELECT      JOB, AVG (SAL)
FROM        EMP
WHERE       JOB! = 'MANAGER'
GROUP BY    JOB;
```

Result:

JOB	AVG (SAL)
ANALYST	3000
CLERK	1037.5
PRESIDENT	5000
SALSMAN	1400

GROUPs within GROUPs:
It is also possible to use GROUP BY clause to provide results for groups within groups.

Example No. 04:
Write a query that displays average monthly salary for each job type in department.

```
SELECT      DEPTNO, JOB, AVG (SAL)
FROM        EMP
GROUP BY    DEPTNO, JOB;
```

Result:

DEPTNO	JOB	AVG (SAL)
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
20	ANALYST	3000
20	CLERK	950
20	MANAGER	2975
30	CLERK	950
30	MANAGER	2850
30	SALESMAN	1400

Joining:
Joining is used to combine the rows from multiple tables. A Join creates a temporary table with all retrieved columns from the tables specified in the Join.

Example:

```
SELECT * FROM A, B;
```

In the above examples, A and B are two tables. It will create a temporary table with all rows from Cartesian product of table A and B. If table A contains 5 rows and table B contains 10 rows, the above statement will retrieve 5 x 10 = 50 rows. It may return duplicate or matching rows. A Join requires horizontal filtering to display desired rows.

The SELECT statement can be written as follows:

SELECT * FROM A, B WHERE X = Y;

Here, X and Y are two columns from table A and B. Table A and B must be joined with a common column between them. The tables must have at least one matching column in order to be joined. A join between two tables should have at least one join condition between them.

Types of Joining:

There are basically three different types of Joins:

1. Simple Join
2. Self-Join
3. Outer Join

1. Simple Join:

Simple Join is the most common type of Join. It retrieves rows from two tables. These tables should have a common column or set of columns that can be logically related. It is further classified into Equi-Join and Non-Equi-Join.

Equi-Join:

A type of Join that is based on equalities is called Equi-Join.

The syntax of Equi-Join is as follows:

```
SELECT    table1.column, table2.column
FROM      table1, table2
WHERE     table1.column = table2.colmumn2;
```

The condition in WHERE clause specifies how the tables are joined.

Example:

Write a query that displays EMPNO, ENAME, DEPTNO, DNAME, LOC from EMP and DEPT tables.

```
SELECT    EMP.EMPNO, EMP.ENAME, EMP.DEPTNO, DEPT.DNAME, DEPT.LOC
FROM      EMP, DEPT
WHERE     EMP.DEPTNO = DEPT.DEPTNO;
```

Result:

EMPNO	ENAME	DEPTNO	DNAME	LOC
7369	SMITH	20	RESEARCH	DALLAS
7499	ALLEN	30	SALES	CHICAGO
7521	WARD	30	SALES	CHICAGO
7566	JONES	20	RESEARCH	DALLAS
7654	MARTIN	30	SALES	CHICAGO
7698	BLAKE	30	SALES	CHICAGO
7782	CLARK	10	ACCOUNTING	NEW YORK
7788	SCOTT	20	RESEARCH	DALLAS
7839	KING	10	ACCOUNTING	NEW YORK
7844	TURNER	30	SALES	CHICAGO
7876	ADANS	20	RESEARCH	DALLAS
7900	JANES	30	SALES	CHICGO
7902	FORD	20	RESEARCH	DALLAS
7934	MILLER	10	ACCOUNTING	NEW YORK

In the above example, the statement EMP.DEPTNO = DEPT.DEPTNO performs the join operation. It retrieves rows from both tables if both have the same DEPTNO as specified in the WHERE clause. The WHERE clause used = operator to perform a join, it is an Equi-join.

The column names are prefixed by table names because both tables have same column name DEPTNO. The tables names distinguish between them. If column names are unique, it is not necessary tables names.

Non-Equi-Join:

A non equi-join specifies the relationship between columns of different tables by using relational (>, <, =, >=, <=, <>) other than =. The following example is illustrating non-equi join.

Example:

Write a query that displays ENAME, SAL, and GRADE from EMP table and GRADE from SALGRAGE table of those employee whose SAL of EMP table is between LOSAL and HISAL of SALGRADE table.

```
SELECT     EMP.ENAME, EMP.SAL, SALGRADE.LOSAL AND SALGRADE.HISAL;
FROM       EMP, SALGRADE
WHERE      EMP.SAL BETWEEN SALGRADE.LOSAL AND SALGRADE.HISAL;
```

Result:

ENAME	SAL	GRADE
SMITH	800	1
ADAMS	1100	1
JAMES	950	1
WARD	1250	2
MARTIN	1250	2
MILLER	1300	2
ALLEN	1600	3
TURNER	1500	3
JONES	2975	4
BLAKE	2850	4
CLARK	2450	4
SCOTT	3000	4
FORD	3000	4
KING	5000	5

2. Self-Join:

Self-Join is a type of join in which a table is joined with itself.

Example:

Write a query that displays EMPNO, ENAME, of employee along with their MGR.

```
SELECT    E.EMPNO, E.ENAME, E.MGR, M.ENAME, MANAGER
FROM      EMP E, EMP M
WHERE     E.MGR = M. EMPNO;
```

Result:

EMPNO	**ENAME**	**MGR**	**MANAGER**
7369	SMITH	7902	FORD
7499	ALLEN	7698	BLAKE
7521	WARD	7698	BLAKE
7566	JONES	7839	KING
7654	MARTIN	7698	BLAKE
7698	BLAKE	7839	KING
7782	CLARK	7839	KING
7788	SCOTT	7566	JONES
7844	TURNER	7698	BLAKE
7876	ADAMS	7788	SCOTT
7988	JAMES	7698	BLAKE
7902	FORD	7566	JONES
7934	MILLER	7782	CLARK

3. Outer Join:

The outer join extends the result of a simple join. It returns all rows returned by simple join as well as those rows from one table that don't match any row from other table. The symbol (+) represents outer join.

Example:

```
SELECT    EMP.EMPNO, EMP.ENAME, EMP.DEPTNO, DEPT.DNAME, DEPT.LOC
FROM      EMP, DEPT
WHERE     EMP. DEPTNO (+) = DEPT.DEPTNO;
```

Result:

EMPNO	ENAME	DEPTNO	DNAME	LOC
----------	----------	-----------	-------------------	------
7782	CLARK	10	ACCOUNTING	NEW YORK
7839	KING	10	ACCOUNTING	NEW YORK
7934	MILLER	10	ACCOUNTING	NEW YORK
7369	SMITH	20	RESEARCH	DALLAS
7876	ADAMS	20	RESEARCH	DALLAS
7902	FORD	20	RESEARCH	DALLAS
7788	SCOTT	20	RESEARCH	DALLAS
7566	JONES	20	RESEARCH	DALLAS
7499	ALLEN	30	SALES	CHICAGO
7698	BLAKE	30	SALES	CHICAGO
7654	MARTIN	30	SALES	CHICAGO
7900	JAMES	30	SALES	CHICAGO
7844	TURNER	30	SALES	CHICAGO
7521	WARD	30	SALES	CHICAGO
			OPERATIONS	BOSTON

The above example retrieves rows form DEPT table that do not have any matching records in EMP table because of the presence of an outer join (+) operator. Outer Join symbol (+) is used after EMP.DEPTNO in WHERE clause. It is always placed on the side which has the data deficiency.

Week No. 16: Data Definition Language and Data Manipulation Language

Data Definition language (DDL) in DBMS:

Data Definition Language can be defined as a standard for commands through which data structures are defined. It is a computer language that is used for creating and modifying structure of the database objects, such as schemas, tables, views, indexes, etc. Additionally, it assists in storing metadata details in the database.

Some of the common Data Definition Language commands are:

1. CREATE
2. ALTER
3. DROP
4. TRUNCATE

1. CREATE:

The CREATE TABLE statement is used to create a new table. Its syntax is as follows:

```
CREATE TABLE table_name
(Column1 datatype, Column2 datatype, Column3 datatype, …….. ColumnN datatype);
```

The Data Type specifies the type of data to be stored in the column. SQL provides the following common Data Types:

Data Type	Description
Char (size)	It is a fixed length character string. The size is specified in parenthesis. The maximum size is 255 bytes.
Varchar (size)	It is a variable length character string. The size is specified in parenthesis.
Number (size)	It is a number value. The maximum number of column digits is specified in parenthesis.
Date	It is the date value.
Number (size, d)	It is number value. The size indicates the maximum number of digits and d indicates the maximum number decimal points.

Example:

```
CREATE TABLE EMP
(EMPNO       NUMBER (4),
 ENAME       VARCHAR2 (30),
 JOB         VARCHAR2 (15),
 MGR         NUMBER (4),
 SAL         NUMBER (7, 2),
 COMM        NUMBER (7, 2),
 DEPTNO      NUMBER (4));
```

2. ALTER:

The ALTER TABLE statement is used to add or drop columns in an existing table. It is also used to change the data type of an existing column of the table.

The General syntaxes of the ALTER command is mentioned below:

ADDING a New Column:
The syntax for adding a new column to an existing table is as follows:

```
ALTER TABLE table_name
ADD column_name datatype;
```

Example:
Write a query to add a column CITY in EMP table.

```
ALTER TABLE EMP
ADD   CITY VARCHAR (30);
```

RENAMING a Table:
The syntax to rename an existing table name is as follows:

```
ALTER TABLE table_name
RENAME To new_table_name;
```

Example:
Write a query to rename the existing table name (EMP) to EMPLOYEE.

```
ALTER TABLE EMP
RENAME To  EMPLOYEE;
```

MODIFYING a Column:
The syntax to modify the data type of an existing column of the table is as follows:

```
ALTER TABLE table_name
MODIFY column_name data type;
```

Example:
Write a query to modify the data type of CITY column to CHAR in EMP table.

```
ALTER TABLE EMP
MODIFY        (CITY CHAR (25));
```

DELETING a Column:
The syntax to delete an existing column from the table is as follows:

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

Example:
Write a query to drop ENAME column in EMP table.

```
ALTER TABLE EMP
DROP  COLUMN ENAME;
```

3. DROP:
The DROP statement is used to delete a table along with table structure, attributes and indexes.

The General syntax of Drop command is mentioned below:

DROP TABLE table_name;

Example:
Write a query to delete EMP table.

DROP TABLE EMP;

4. TRUNCATE:
By using Truncate command, users can remove table content, but structure of the table is kept. In simple language, it removes all the records from the table structure. Users can't remove data partially through this command. In addition to this, every space allocated for the data is removed by Truncate command.

The syntax of the Truncate command is mentioned below:

TRUNCATE TABLE table_name;

Example:
Write a query to delete all data in EMP table.

TRUNCATE TABLE EMP;

Data Manipulation Language (DML) in DBMS:
Data manipulation language (DML) provides operations that handle user requests, offering a way to access and manipulate the data that users store within a database. Its common functions include inserting, updating and retrieving data from the database.

Here is a list of DML statements:

1. INSERT
2. UPDATE
3. DELETE

1. INSERT:
The INSERT INTO statement is used to insert new rows in a table.

The syntax of this statement is as follows:

INSERT INTO table_name
VALUES (value1, value2,………..);

Example:

INSERT INTO DEPT
VALUES (50, 'EDUCATION', 'LAHORE');

The column names can also be specified for which data is to be inserted:

```
INSERT INTO table_name (column1, column2,…….)
VALUES (value1, value2,…….);
```

Example:

```
INSERT INTO DEPT (DEPTNO, DNAME)
VALUES (60, ‘MIS’);
```

2. UPDATE:
The UPDATE statement is used to modify the existing data in a table.

The syntax of using this statement is as follows:

```
UPDATE table_name
SET column_name = new_value
WHERE column_name = some_value;
```

UPDATE One Column in a Row:
The following statement will modify DEPTNO of SMITH:

```
UPDATE EMP SET DEPTNO = 70
WHERE ENAME = ‘SMITH’;
```

UPDATE Several Columns in a Row:
The following statement will modify JOB and ENAME of SMITH:

```
UPDATE EMP
SET JOB = ‘SALESMAN’, ENAME = ‘CLERK’
WHERE ENAME = ‘SMITH’;
```

3. DELETE:
The DELETE statement is used to delete rows in a table.

The syntax of this statement is as follows:

```
DELETE FROM table_name
WHERE column_name = some_value;
```

DELETE a Row:
The following example delete SMITH:

```
DELETE FROM EMP WHERE ENAME = ‘SMITH’;
```

DELETE All Rows:
It is possible to delete all rows in a table without deleting the table. The following statement will delete all data in the table but table structure, attributes, and indexes will not be deleted.

DELETE FROM table_name; **OR** DELETE * FROM table_name;

Example:

DELETE FROM EMP; **OR** DELETE * FROM EMP;