T.E. Computer

Database Management Systems

Unit-III

By

Prof. Dr. K.P. Adhiya

SSBT's COET, Bambhori-Jalgaon

Acknowledgment

1. Abraham Silberschatz, Henry F. Korth, S. Sudarshan, “Database System Concepts”, 6th Edition, McGraw-Hill Hill Education.
2. Abraham Silberschatz, Henry F. Korth, S. Sudarshan, “Database System Concepts”, 4th Edition, McGraw-Hill Hill Education.
3. Ramez Elmasri and Shamkant B. Navathe “Fundamentals of Database Systems”, 5th Edition, Pearson.
4. Express Learning, “Database Management Systems”, ITL Education Solutions Limited.
5. Archana Verma, “Database Management Systems”, GenNext Publication.
6. Dr. Rajiv Chopra, “Database Management Systems (DBMS) – A Practical Approach”, 5th Edition, S. Chand Technical
7. Tanmay Kasbe, “Database Management System Concepts – A Practical Approach”, First Edition, Educreation Publishing.
8. Mahesh Mali, “Database Management Systems”, Edition 2019, TechKnowledge Publications.
9. Rajendra Prasad Mahapatra, Govind Verma, “Database Management System”, Khanna Publishing.
10. Malay K. Pakhira, “Database Management System”, Eastern Economy Edition, PHI.
11. Sarika Gupta, Gaurav Gupta, “Database Management System”, Khanna Book Publishing Edition.
12. Riktesh Srivastava, Rajita Srivastava, “Relational Database Management System”, New Age International Publishers.
13. Peter Rob, Carlos Coronel, “Database System Concepts’, Cenage Learning, India Edition
14. Bipin C. Desai, “An Introduction to Database Systems”, Galgotia Publications.
15. G.K. Gupta, “Database Management Systems”, McGraw Hill Education.
16. Shio Kumar Singh, “Database Systems – Concepts, Design and Applications”, 2nd Edition, PEARSON.
17. S.D.Joshi, “Database Management System”, Tech-Max Publication.
18. R. Ramkrishnan , J. Gehrke, "Database Management Systems", 3rd Edition, McGraw-Hill
19. C. J. Date, “Introduction to Database Management Systems”, 8th Edition, Pearson
20. Atul Kahate, “Introduction to Database Management System”, 3rd Edition, Pearson.
21. Bharat Lohiya, “Database Systems”, Tenth Edition, Aditya Publication, Amravati.
22. Vijay Krishna Pallaw, “Database Management System”, 2nd, Asian Books Pvt. Ltd.
23. Database Management Systems, Database Management Systems.
24. Mrs. Jyoti G. Mante (Khurpade), Mrs. Smita M. Dandge, “Database Mangement System”, Nirali Prakashan.
25. Step by Step Database Systems (DBMS), Shiv Krupa Publications, Akola

26. Mrs. Sheetal Gujar –Takale, Mr. Sahil K. Shah, "Database Management System", Nirali Prakashan.
27. Mrs. Jyoti G. Mante (Khurpade), U.S. Shirshetti, M.V. Salvi, K.S. Sakure, "Relational Database Management System", Nirali Prakashan.
28. Seema Kedar, Rakesh Shirsath, "Database Management Systems", Technical Publications.
29. Pankaj B. Brahmankar, "Database Management Systems", Tech-Max Publications, Pune.
30. Imran Saeed, Tasleem Mustafa, Tariq Mahmood, Ahsan Raza Sattar, "A Fundamental Study of Database Management Systems", 3rd Edition, IT Series Publication.
31. Database Management Systems Lecture Notes, Malla Reddy College of Engineering and Technology, Secunderabad.
32. Dr. Satinder Bal Gupta, Aditya Mittal, "Introduction to Database Management System, University Science Press.
33. E-Notes BCS 41/ BCA 41 on "Database Management System", Thiruvalluvar University.
34. Bighnaraj Naik, Digital Notes on "Relational Database Management System", VSSUT, Burla.
35. Viren Sir, Relational database Management System", Adarsh Institute of Technolgoyt (Poly), VITA.
36. Sitansu S. Mitra, "Principles of Relational Database Systems", Prentice Hall.
37. Neeraj Sharma, Liviu Perniu, Raul F. Chong, Abhishek Iyer, Chaitali Nandan, Adi-Cristina Mitea, Mallarswami Nonvinkere, Mirela Danubianu, "Database Fundamentals", First Edition, DB2 On Campus Book Series.
38. Database Management System, Vidyavahini First Grade College, Tumkur.
39. Bhavna Sangamnerkar, Revised by: Shiv Kishor Sharma, "Database Management System", Think Tanks Biyani Group of Colleges.
40. Tibor Radvanyi, "Database Management Systems".
41. Ramon A. Mata-Toledo, Pauline K. Cushman, "Fundamentals of Relational Databases", Schaum's Outlies.

Acknowledgment

Web Resources

https://www.sqlshack.com/sql-union-vs-union-all-in-sql-server/
https://www.sqltutorial.org/sql-intersect/
https://gokhanatil.com/2010/10/minus-and-intersect-in-mysql.html
https://www.sqltutorial.org/sql-minus/
https://www.w3schools.com/sql/sql_in.asp
https://www.geeksforgeeks.org/
https://www.w3resource.com/sql/joins/natural-join.php
https://www.guru99.com/joins-sql-left-right.html
https://www.w3schools.com/sql/sql_join.asp
https://www.guru99.com/joins-sql-left-right.html
http://www.sql-join.com/
https://www.geeksforgeeks.org/sql-join-set-1-inner-left-right-and-full-joins/
https://www.tutorialspoint.com/sql/sql-using-joins.htm
https://www.tutorialspoint.com/difference-between-views-and-materialized-views-in-sql
https://tutorialink.com/dbms/procedures-and-functions.dbms
https://docs.oracle.com/cd/E19078-01/mysql/mysql-refman-5.0/stored-programs-views.html
https://www.mysqltutorial.org/mysql-stored-procedure/mysql-while-loop/
https://www.mysqltutorial.org/mysql-if-statement
https://www.javatpoint.com/mysql-trigger
https://www.javatpoint.com/mysql-trigger
https://www.w3resource.com/mysql/mysql-triggers.php
https://link.springer.com/content/pdf/bbm%3A978-1-4302-0867-9%2F1.pdf
https://www.w3resource.com/sql/sql-basic/codd-12-rule-relation.php

Unit-III

(Introduction to the Relational Model)

➢ Structure of Relational Databases

- Nowadays, the relational model is the primary data model for commercial processing applications, because of its simplicity, which eases job of the programmer.
- A relational database consists of a collection of **tables**. Each table is assigned with a unique name.
- Table is also called as **relation**. (In E-R terminology, it is entity).
- For example, consider the **instructor table**, shown in figure 3.1. It stores the information of various instructors.

ID	name	dept_name	salary
100	George	Computer	110000
101	Krishna	Computer	95000
102	Uday	IT	105000
103	Swapnil	ETC	112000
104	Soumitra	Mechanical	115000
105	Manas	ETC	98000
106	Pawan	Electrical	106000
107	Navin	Mechanical	101000
108	Devendra	ETC	97000
109	Parth	Electrical	103000
110	Nilesh	IT	70000

Figure 3.1:- The instructor relation

- The table has four columns – ID, name, dept_name, salary
- Each row of the table is called as **record**. It is also called as **tuple**.
- The instructor table contains 11 records.

- Figure 3.2 shows the **course table**. It stores information such as- course_id, title, dept_name and credit.

course_id	Title	dept_name	credits
ETC-101	Analog Electronics	ETC	3
ETC-102	Microprocessor	ETC	4
ETC-103	Advanced Microprocessor	ETC	3
Mech-101	Engineering Drawing	Mechanical	3
Mech-102	Machine Design	Mechanical	4
Comp-101	Compiler Design	Computer	4
Comp-102	DBMS	Computer	4
Comp-103	Data Mining	Computer	4
Elect-101	Power Electronics	Electrical	3
Elect-102	Electromagnetic Field	Electrical	4
IT-101	Computer Network	IT	4
IT-102	Network Security	IT	3

Figure 3.2:- The course relation

- Figure 3.3 shows a third table, prereq, which stores the prerequisite courses for each course. This table has two columns – course_id and prereq_id.

course_id	prereq_id
IT-102	IT-101
Mech-102	Mech-101
Comp-103	Comp-102
ETC-103	ETC-102

Figure 3.3:- The prereq relation

- In general, a row a table represents a relationship among a set of values. Since a table is a collection of such relationships, hence it is called as **relation.**

- Thus, in the relational model the term **relation** is used to refer to a table, while the term **tuple** is used to refer to a row. Similarly, the term **attribute** refers to a column of a table.
- **Relation Instance**:- It refers to the specific instance of a relation, i.e. containing a specific set of rows. E.g. figure 3.1 shows instance of a instructor relation, which has 11 tuples and 4 attributes (columns). The table gives the information of 11 instructors.
- The order in which tuples appear in a relation is irrelevant. Thus, whether the tuples of a relation are listed in sorted order, as in Figure 3.1, or are unsorted, does not matter.
- **Domain**: - It is a collection of all possible values (or permitted values) of each attribute of a relation. Thus, the domain of the *salary* attribute of the *instructor* relation is the set of all possible salary values, while the domain of the *name* attribute is the set of all possible instructor names.
- The null value is a special character value that signifies that the value is unknown or does not exist. E.g. credits attribute of course relation may have null value, if credits is not applicable to some subject. (e.g. Internship in our syllabus).
- **Degree of a relation**: - The number of attributes in a relation is known as degree of relation. E.g. the degree for instructor relation is four.
- **Cardinality of a relation**:- The number of tuples in a relation is called as cardinality of a relation. E.g. cardinality of instructor relation is 11.
- **Relational DBMS (RDBMS) terminologies**:-

Formal Relational Terms	Informal Equivalent Terms
Relation	Table
Tuple	Record, Row
Cardinality	Number of rows
Attribute	Column, field
Degree	Number of columns
Domain	Set of possible (permitted) values
Primary key	Unique identifier

- Properties of relation:-
  Generally, a relation has following properties-
  - A relation is a table with columns and rows
  - A relation has a name that is distinct from all other relation names in the given database.
  - Each cell of relation contains exactly one atomic (single) value.
  - Each attribute has a distinct name.
  - The values of an attribute are all from the same domain.
  - Each tuple is distinct; i.e. there are no duplicate tuples.
  - The order of attributes has no significance.
  - The order of tuples has no significance, theoretically.

➢ Database Schema

- **Database schema**: - It is logical design of the database. The database schema is the overall structure of the database, which specifies the structure of each relation including its attributes (considering relational model). It does not include the actual records of the relation.
- **Database instance**: - It is a snapshot of the data in the database at a given instant of time.
- **Relation schema**: - It consists of list of attributes of the table. It consists of relation name R and a set of attributes A_1, A_2, A_3,…A_n. It is represented by R (A_1, A_2,…A_n) and is used to describe a relation R.
- **Relation instance**: - It is a snapshot of the data in a relation at a given instant of time.
- Consider the relation instructor from figure 3.1. The schema for this relation is –

  instructor (ID, name, dept_name, salary)

  It can be also represented as –

  instructor

ID	name	dept_name	salary

  Similarly, consider the following relation department –

dept_name	building	budget
Computer	Building No. 1	1000000
IT	Building No. 2	500000
ETC	Building No. 1	800000
Mechanical	Building No. 3	900000
Electrical	Building No. 2	600000

Figure 3.4:- The department relation

The schema for this department relation is –

department (dept_name, building, budget)

- So here, in the college example, there are four relation schemas to form the relational database schema –

  instructor (ID, name, dept_name, salary)
  course (course_id, title, dept_name, credits)
  prereq (course_id, prereq_id)
  department (dept_name, building, budget)

➢ Keys

There must be a way to specify how tuples within a given relations are distinguished. This is expressed in terms of their attributes. No two tuples in a relation are allowed to have exactly the same value for all attributes. There are different types of keys –

- **Superkey** :- A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation. For example, the ID attribute of the relation *instructor* is sufficient to distinguish one tuple from another. Thus, ID is a superkey.

  The combination of ID and name is also a superkey for instructor relation. But, the name attribute of instructor, is not a superkey, because several instructors might have the same name.
- **Candidate key** :- Generally all the attributes of a superkey are not required to identify each tuple uniquely in a relation (i.e. a superkey may contain **extraneous** attributes). Instead, only a subset of attributes of the superkey is sufficient to uniquely identify each tuple. Such a minimal set of attributes is a **candidate key** (also known as irreducible superkey).

The combination of {ID, name} → superkey

The combination of {ID, name} → not a candidate key, since the attribute ID alone is a candidate key.

- **Primary Key** :- There can be more than one candidate key for a relation. So the candidate key that is chosen by the database designer to uniquely identify the tuple in a given relation is called as **primary key**. Other candidate keys that are not chosen as primary key are called as **alternate keys.**

  The alone {ID} → candidate key

  The Combination of {name, dept_name} → candidate key, assuming that in a department no two instructors have exactly the same name.

  Now, database designer choose ID as a primary key.

  Primary key must be chosen with care. It should be chosen such that its attributes values are never or rarely changed.

  Generally, the primary key attributes of a relation schema are listed before the other attributes; for example, the *ID* attribute of *instructor* is listed first, since it is the primary key. Primary key attributes are also underlined. E.g. ID

- **Foreign Key** :- Attribute of one relation can be accessed in another relation by enforcing a link between the attributes of two relations. This can be done by defining the attribute of one relation as the **foreign key** that refers to the primary key of another relation.

  The foreign key belongs to the **referencing** relation and the primary key belongs to the **referenced** relation.

  In a department relation (figure 3.4) →dept_name is primary key

  In a instructor relation (figure 3.1) → dept_name is a foreign key. So the instructor relation is a referencing relation and department relation is a referenced relation.

  Now, we will rewrite the four relation schemas with primary indicated by underline.

  instructor (ID, name, dept_name, salary)
  course (course_id, title, dept_name, credits)
  prereq (course_id, prereq_id)
  department (dept_name, building, budget)

➢ Schema Diagrams

- A database schema, along with primary key and foreign key dependencies, can be shown by schema diagrams.
- Some authors didn't make any difference between the schema diagrams and database schema.
- Primary key attributes in a relation are underlined.
- Foreign key dependencies appear as arrows from the foreign key attributes of the **referencing relation** to the primary key of the **referenced relation**.
- Consider the College database schema as follow –

COURSE

C-ID	C-Name

STUDENT

S-ID	S-Name	S-Address

STUDENT-PHONE

Stu-ID	S-Phone

DEPARTMENT

D-ID	D-NAME	Prof-ID

PROFESSOR

P-ID	P-Name	Dept-ID	P-Salary	Head-ID

Figure 3.5

- Note, in a relation STUDENT-PHONE, the combination of {Stu-ID, S-Phone} is a primary key. (Note – Some SQL may not support the name of attribute as Stu-ID, so use Stu_ID).
- Foreign key Dept-ID of PROFESSOR table refers to the primary key D-ID of DEPARTMENT table.
- The schema diagram for above database schema can be as follow –

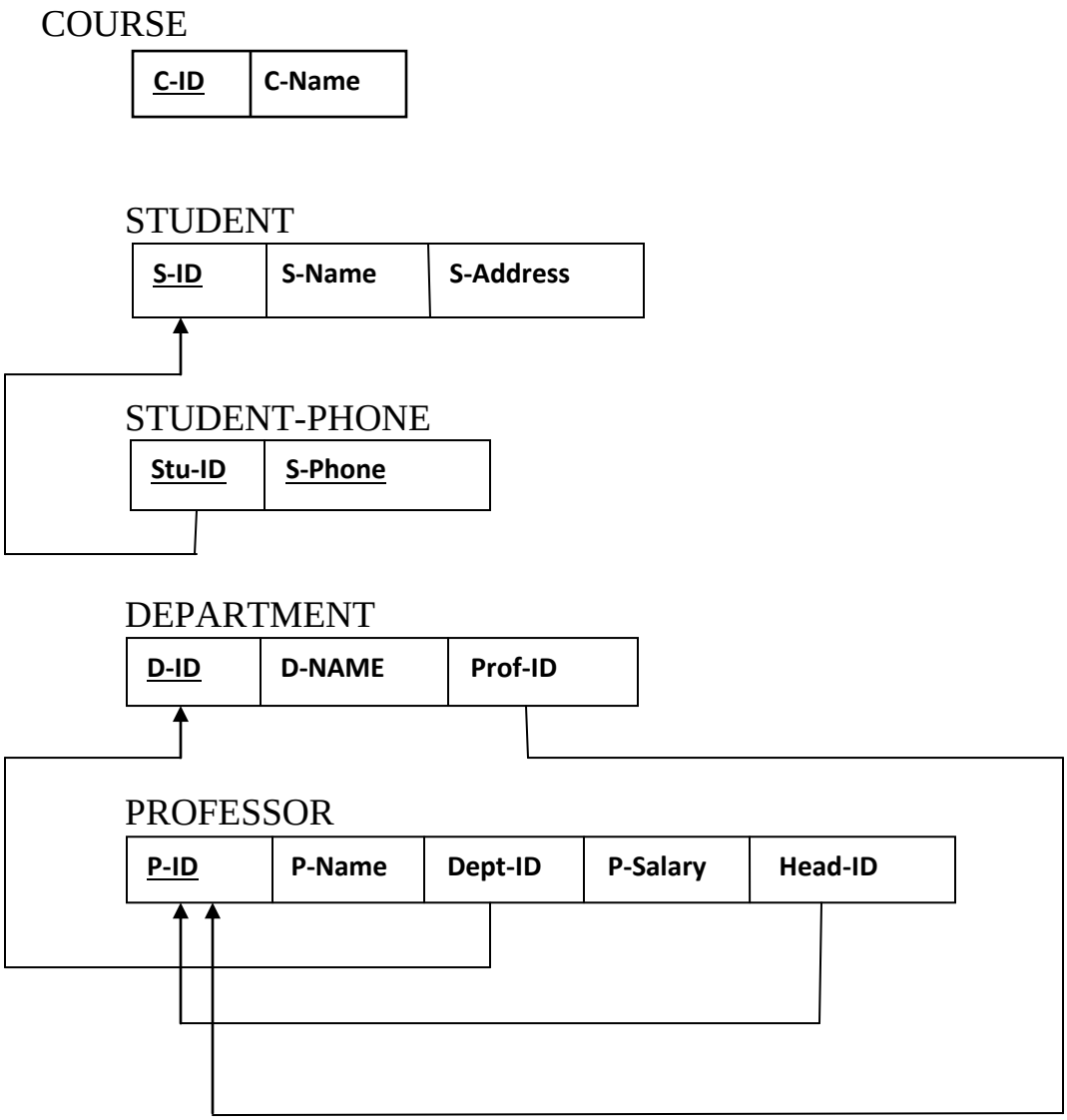


Figure 3.6

➢ Codd's Rules

Dr. Edgar F Codd formulated 13 rules including zero rule for relational DBMS. Based on these rules, we can decide a DBMS to be an RDBMS (Relational Database Management System) or not. The very limited number of commercial product follows all rules. The Codd's rules are stated as follow:-

- **Rule Zero - Foundation Rule: -** The system must qualify as relational, as a database, and as a management system. The RDBMS

must manage every aspect of the database entirely thorough its relational capabilities.

- **Rule 1- Information Rule:** - All information in the relational database is represented in exactly one and only one way—by values in tables (i.e. relations).
- **Rule 2 – Guaranteed Access Rule: -** Each and every datum (atomic value) is guaranteed to be logically accessible by a combination of table name, primary key value (to find the row), and column name.
- **Rule 3 – Systematic Treatment of NULL values: -** NULL values are supported in the fully relational RDBMS for representing missing information in a systematic way, independent of data type.
- **Rule 4 – Active online Catalog**: - The RDBMS should maintain online catalog known as data dictionary, to keep the track of current state of the database. This data dictionary will be accessed by authorized users using the same query language which they apply to access the regular data.
- **Rule 5 – Comprehensive Data Sublanguage (DSL) Rule: -** A relational system may support several languages. However, there must be at least one language which will support at least all of the followings –
    - Data definition
    - View definition
    - Data manipulation operations
    - Integrity constraints
    - Authorization
    - Transaction management operations
- **Rule 6 – View Updation Rule: -** All views that are theoretically updateable are also updateable by the system.
- **Rule 7 – High-Level Insert, Update and Delete Rule: -** The RDBMS must be capable of retrieving the relational data as well as must be able to do insertion, updation and deletion operations. It should be able to do these operations on more than one row also.

- **Rule 8 – Physical Data Independence Rule: -** Application programs must remain unchanged whenever any changes are made in either storage representation or access methods.
- **Rule 9 – Logical Data Independence Rule: -** Any modification in conceptual schema should not force the modifications in application programs.
- **Rule 10 – Integrity Independence Rule: -** Integrity constraints must be specified independent of application programs. They must be storable in the data dictionary and not in application programs.
- **Rule 11 – Distribution Independence Rule:-** There should not be any problem to the users for accessing the data, whether the data is physically centralized or distributed. This emphasizes the fact that the system should look like a centralized system to the user, even if it is distributed across.
- **Rule 12 – Non-Subversion Rule: -** If RDBMS has a low-level (single-record-at-a-time) interface, then it must not bypass the integrity rules or constraints of the relational language. This rule requires that alternate methods of accessing the data are not able to bypass integrity constraints, which means that users can't violate the rules of the database in any way.

Unit-III

(Introduction to SQL)

➢ Overview of the SQL Query language

- SQL (Structured query Language) has established itself as the standard relational database language.
- It is a language that can be used for retrieval and management of data stored in relational database.
- It is a non-procedural language as it specifies what is to be retrieved rather than how to retrieve it.
- The SQL language has several parts:-
  - **Data-definition language** (DDL). The DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas. E.g. Create table, Alter table, Drop table etc.
  - **Data-manipulation language** (DML). The DML provides the ability to query information from the database and to insert tuples, delete tuples, modify tuples.
  - **Integrity**. The DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy.
  - **View definition**. The DDL includes commands for defining views.
  - **Transaction control**. SQL includes commands for specifying the beginning and ending of transactions.
  - **Embedded SQL** and **dynamic SQL**. Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, and Java.
  - **Authorization**. The DDL includes commands for specifying access rights to relations and views.
- Features of SQL:-
  - It is English like language. E.g. it uses words like select, insert, delete etc.
  - It is a non procedural language.
  - SQL commands are not case sensitive.

- SQL can be used by range of users.
- SQL can proceed with a set of records.
- SQL provides commands for various tasks including –querying the data, inserting/updating/deleting the rows, controlling access to the database.

➢ SQL Data Definition

- The set of relations in a database must be specified to the system by means of a **Data-Definition Language (DDL).**

✓ **Basic Types (i.e. Basic Data Types) –** The SQL supports a variety of built-in data types, including :-

- **char** (n) or **character** (n) **:** used to represent data as a fixed-length string of character of size n.
- **varchar** (*n*) or **character varying** (n) : used to represent a variable-length character string with user-specified maximum length *n*.
- **int** or **integer:** used to represent data as a number without a decimal point.
- **smallint**: A small integer. It is used to represent data as a number without a decimal point. Default size is usually smaller that int.
- **numeric** (*p, d*): used to represent a data as floating point number with user-specified precision. The number consists of *p* digits (plus a sign), and *d* of the *p* digits are to the right of the decimal point. Thus, **numeric**(3,1) indicates that the value can be 43.5
- **real, double precision**: Floating-point and double-precision floating-point numbers with machine-dependent precision.
- **float**(*n*): A floating-point number, with precision of at least *n* digits.
- **date** and **time :** used to represent data as date or time.
- **timestamp :** used to represent data as both date and time.

Each type may include a special value called the **null** value.

Use of **varchar** type is recommended instead of char type.

✓ Basic Schema Definition

The SQL relation can be defined by using the **create table** command.

Consider the figure 3.4; the following command creates a table (i.e. relation) department –

```
create table department
        ( dept_name   varchar (20),
          building      varchar (15),
          budget        numeric (12,2),
          primary key (dept_name));
```

The general form of the **create table** command is:

```
create table table_name
        ( attribute A1      data type D1,
          attribute A2      data type D2,
          . . . ,
          attribute An      data type Dn,

        < integrity-constraint1 >,
          . . . ,
        < integrity-constraintk >);
```

Integrity constraints are optional. The semicolon shown at the end of the create table statements, as well as at the end of other SQL statements, is optional in many SQL implementations.

The SQL supports a number of different **integrity constraints**. Some of the constraints are as follow:

- **Primary key** :- The primary key attributes are required to be **nonnull and unique** ; that is, no tuple can have a null value for a primary key attribute, and the attributes forming the primary key cannot have duplicate values. The primary-key specification is optional, but it is good idea to specify a primary key for each relation.
- **Foreign key**: - The value of foreign key in the referencing relation must exist in the primary key attribute of the referenced relation.

- **Not null :-** The not null constraint on an attribute specifies that the null value is not allowed for that attribute.
- **Unique Constarint:-** Refer Page No. 73
- **Check constraint:-** Refer Page No. 74
- **Domain Constraint:-** Refer Page No. 73

- **create table command** -

Consider the figures 3.5 and 3.6 –

```
create table COURSE
        ( C_ID          varchar (5),
          C_Name        varchar (20) not null,
          primary key   (C_ID));

create table STUDENT
        ( S_ID          varchar (5),
          S_Name        varchar (20) not null,
          S_Address     varchar (30)
          primary key   (S-ID));

create table STUDENT-PHONE
        ( Stu_ID        varchar (5),
          S_Phone       varchar (10),
          primary key (Stu_ID, S_Phone),
          foreign key   (Stu_ID) references
                        STUDENT (S_ID) );

create table DEPARTMENT
        ( D_ID          varchar (5),
          D_NAME        varchar (20) not null,
          Prof_ID       varchar (5) not null,
          primary key (D_ID),
          foreign key   (Prof_ID) references
                        PROFESSOR (P_ID));
```

```
create table PROFESSOR
        ( P_ID          varchar (5),
          P_NAME    varchar (20) not null,
          Dept_ID      varchar (5) not null,
          P_Salary      numeric (8, 2),
          Head_ID       varchar (5),
          primary key (P_ID)
          foreign key (Dept_ID) references
                        DEPARTMENT (D_ID)
          foreign key (Head_ID) references
                        PROFESSOR (P_ID));
```

Figure 3.7: SQL data definition

- **insert into command -**

  A newly created relation is empty initially. We can use the **insert** command (it is DML command) to load data into the database. E.g.

```
insert into STUDENT
        values ('11001', 'BUNTY', 'JALGAON');
```

  The values are specified in the order in which the corresponding attributes are listed in the relation schema.

- **delete command -**

  We can use the **delete** command to delete tuples (records) from a relation. E.g. the command –

```
delete from student ;
```

  This command will delete all tuples from the student relation.

- **drop table command -**

  To remove a relation from the database, we can use **drop table** command. E.g.

drop table student;

The **drop table** command deletes not only all tuples of a relation, but also the schema of a relation. After dropping the relation using drop table command, no tuples can be inserted into that relation.

- **alter table command -**

We can use **alter table** command to add attributes to an existing relation. The form of the alter table command is

alter table r **add** A D;

where, r →name of an existing relation

A →name of an attribute to be added

D → type of added attribute

We can drop attributes from a relation by the command –

alter table r **drop** A;

where, r →name of an existing relation

A →name of an attribute to be dropped.

- **rename command :-** The **rename table** command renames one or more tables.

E.g. **rename table** old_table **to** new_table;

Above statement is equivalent to the following statement:-

alter table old_table **rename** new_table;

Following statement renames multiple tables:-

rename table old_table1 **to** new_table1,
rename table old_table2 **to** new_table2,
rename table old_table3 **to** new_table3;

➢ Basic Structure of SQL Queries

- The basic structure of SQL query consists of three clauses: **select**, **from**, and **where**.

- The basic structure of SQL query is –

SELECT <column_name>
FROM <table_name>
OR
SELECT <column_name>
FROM <table_name>
WHERE <condition>

✓ Queries on a Single Relation :-

- First you have to create all the tables using **create table** command and then insert the records in the tables using **insert into** command.
- Consider the table (i.e. relation) instructor from figure 3.1 and the records as follow –

ID	name	dept_name	salary
100	George	Computer	110000
101	Krishna	Computer	95000
102	Uday	IT	105000
103	Swapnil	ETC	112000
104	Soumitra	Mechanical	115000
105	Manas	ETC	98000
106	Pawan	Electrical	106000
107	Navin	Mechanical	101000
108	Devendra	ETC	97000
109	Parth	Electrical	103000
110	Nilesh	IT	70000

Figure 3.8:- The **instructor** relation

- Now consider a query → "Find the names of all instructors". So, the SQL query will be-

select name
from instructor;

The result of this query is a relation consisting of a single attribute with the heading **name** as shown in figure 3.9 as follow -

name
George
Krishna
Uday
Swapnil
Soumitra
Manas
Pawan
Navin
Devendra
Parth
Nilesh

Figure 3.9

- Now consider a query → "Find the department names of all instructors". So, the SQL query will be-

```
select dept_name
from instructor;
```

Here, since more than one instructor can belong to a department, a department name could appear more than once in the resultant relation, as shown in figure 3.10 as follow -

dept_name
Computer
Computer
IT
ETC
Mechanical
ETC
Electrical
Mechanical
ETC
Electrical
IT

Figure 3.10

- SQL allows duplicates in relations as well as in the results of SQL expressions. See figure 3.10 and its related query.
- To force the elimination of duplicates, use the keyword **distinct** after **select.** E.g.

  select distinct dept_name
  from instructor;

  then the resultant will be as shown in figure 3.11.

dept_name
Computer
IT
ETC
Mechanical
Electrical

Figure 3.11

- Use of **all** keyword – Consider a query as follow:

  select all dept_name
  from instructor;

  The result of this query will be same as shown in figure 3.10. Hence we can say that, duplicate retention is by default, so generally **all** keyword is not used.
- The **select** clause may also contain arithmetic expressions involving the operators +, −,*, and / operating on constants or attributes of tuples. For example the query :

  select *ID*, *name*, *dept_ name*, *salary* +1000
  from *instructor*;

  Then the resultant relation will be as shown in figure 3.12 (the All attribute values are same except that, attribute salary is increased by 1000.)

ID	**name**	**dept_name**	**salary**
100	George	Computer	111000
101	Krishna	Computer	96000
102	Uday	IT	106000
103	Swapnil	ETC	113000

104	Soumitra	Mechanical	116000
105	Manas	ETC	99000
106	Pawan	Electrical	107000
107	Navin	Mechanical	102000
108	Devendra	ETC	98000
109	Parth	Electrical	104000
110	Nilesh	IT	71000

Figure: - 3.12

- Use of **where** clause: - The **where** clause allows us to select only those rows in the result relation of the **from** clause that satisfy a specified predicate (i.e. condition). Consider the query (for figure 3.8), "Find the names of all instructors in the ETC department who have salary greater than 100000." This query can be written in SQL as:

  select *name*
  from *instructor*
  where dept_name = 'ETC' **and** salary > 100000;

  Then the result will be as shown in figure 3.13

name
Swapnil

Figure 3.13

- SQL allows the use of the logical connectives **and**, **or**, and **not** in the **where** clause. The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <> *(i.e. not equal to)*.

✓ **<u>Queries on Multiple Relations</u>**: **-** Queries often need to access information from multiple relations.

- So, e.g. again consider the following two relations from figure 3.1 and 3.4.

ID	name	dept_name	salary
100	George	Computer	110000
101	Krishna	Computer	95000
102	Uday	IT	105000
103	Swapnil	ETC	112000
104	Soumitra	Mechanical	115000
105	Manas	ETC	98000
106	Pawan	Electrical	106000
107	Navin	Mechanical	101000
108	Devendra	ETC	97000
109	Parth	Electrical	103000
110	Nilesh	IT	70000

Figure 3.14:- The instructor relation

dept_name	building	budget
Computer	Building No. 1	1000000
IT	Building No. 2	500000
ETC	Building No. 1	800000
Mechanical	Building No. 3	900000
Electrical	Building No. 2	600000

Figure 3.15:- The department relation

Now, consider the query → "Retrieve the names of all instructors, along with their department names and department building."
Here you will require two relations to be accessed. So in the required query, list the relation names that need to be accessed in the **from** clause, and specify the **matching condition** in the **where** clause.

select *name, instructor.dept_name, building*
from *instructor, department*
where instructor.dept_name = department.dept_name;

The result of this query will be as follow:-

name	dept_name	building
George	Computer	Building No. 1
Krishna	Computer	Building No. 1
Uday	IT	Building No. 2
Swapnil	ETC	Building No. 1
Soumitra	Mechanical	Building No. 3
Manas	ETC	Building No. 1
Pawan	Electrical	Building No. 2
Navin	Mechanical	Building No. 3
Devendra	ETC	Building No. 1
Parth	Electrical	Building No. 2
Nilesh	IT	Building No. 2

Figure 3.16

- The role of each clause in the case of SQL queries involving multiple relations will be as follow-
  - The **select** clause is used to list the attributes desired in the result of a query.
  - The **from** clause is a list of the relations to be accessed in the evaluation of the query.
  - The **where** clause is a predicate involving attributes of the relation in the **from** clause.

  So, the typical SQL query has the form as –

  select $A_1, A_2, \ldots, A_n$
  from $r_1, r_2, \ldots, r_m$
  where P;

- **Cartesian Product**: - The **from** clause by itself defines a Cartesian Product of the relations listed in the clause. E.g. consider the relations instructor and department from figures 3.14 and 3.15, but in a reduced form (i.e. less number of records), named as instructor_r and department_r.

ID	name	dept_name	salary
100	George	Computer	110000
101	Krishna	Computer	95000
102	Uday	IT	105000

Figure 3.17:- instructor_r relation, in a reduced form

dept_name	building	budget
Computer	Building No. 1	1000000
IT	Building No. 2	500000
ETC	Building No. 1	800000

Figure 3.18:- department_r relation, in a reduced form

The Cartesian product combines tuples from instructor_r and department_r. **Each tuple in instructor_r is combined with each tuple in department_r._** The degree of a new resultant relation will be sum of the degree of both the relations, i.e. 7. The result of this Cartesian product is as shown in following figure 3.19 –

Note: - Overall structure of the resultant relation may depend on query fired.

ID	name	instructor_r. dept_name	salary	department_r .dept_name	building	budget
100	George	Computer	110000	Computer	Building No. 1	1000000
101	Krishna	Computer	95000	Computer	Building No. 1	1000000
102	Uday	IT	105000	Computer	Building No. 1	1000000
100	George	Computer	110000	ETC	Building No. 1	800000
101	Krishna	Computer	95000	ETC	Building No. 1	800000
102	Uday	IT	105000	ETC	Building No. 1	800000
100	George	Computer	110000	IT	Building No. 2	500000
101	Krishna	Computer	95000	IT	Building No. 2	500000
102	Uday	IT	105000	IT	Building No. 2	500000

Figure 3.19

The predicate in the **where** clause is used to restrict the combinations created by the Cartesian product. E.g.

select *name, salary, building*
from *instructor_r, department_r*
where instructor_r.dept_name = department_r.dept_name;

name	**salary**	**building**
George	110000	Building No. 1
Krishna	95000	Building No. 1
Uday	105000	Building No. 2

So, in general the meaning of an SQL query can be understood as follow –

1. Generate a Cartesian product of the relations listed in the **from** clause
2. Apply the predicates specified in the **where** clause on the result of Step 1.
3. For each tuple in the result of Step 2, output the attributes (or results of expressions) specified in the **select** clause.

✓ The Natural Join:-

- The Cartesian product of two relations concatenates each tuple of the first relation with every tuple of the second relation.
- Natural join considers only those pairs of tuples with the same value on those attributes that appear in the schemas of both relations.
- In other words – we can perform Natural Join only if there is at least one common attribute that exists between two relations. Natural join acts on those matching attributes where the values of attributes in both the relations are same. The natural join eliminates the duplication of common attributes that appear in both relations. Therefore common column will appear only once in a resultant relation.

- E.g. the following query –

  select name, salary, budget
  from instructor_r, department_r
  where instructor_r.dept_name = department_r.dept_name;

name	**salary**	**budget**
George	110000	1000000
Krishna	95000	1000000
Uday	105000	500000

- The above query can be written using **natural join** operation in SQL as –

  select name, salary, budget
  from instructor_r
  natural join department_r;

  <u>This query will also generate the same result.</u>

- A from clause in an SQL query can have multiple relations combined using natural join , as shown –

  select A_1, A_2, …..A_n
  from r_1 **natural join** r_2 **natural join** …..**natural join** r_m
  where P;

  More generally, a from clause can be of the form-

  from E_1, E_2,….., E_n

  where E_i can be a single relation or an expression involving natural joins.

➢ Additional Basic Operations

Some additional basic operations that are supported by SQL:

✓ The Rename Operation –

SQL provides a way of renaming the attributes of a result relation. It uses the **as** clause, taking the form:

Old-name **as** new-name

The **as** clause can appear in both the **select** clause and **from** clause.
E.g. if we want attribute name dept_name to be replaced with the name dname, we can write the query as –

select dept_name **as** dname **from** department_r;

The output of this query will be –

dname
Computer
ETC
IT

If the query is given as –

select dept_name **from** department_r;

then the output of the query will be as follow -

dept_name
Computer
ETC
IT

One of the reasons to use rename operation is - to replace a long relation name with a short version that is more convenient to use.
E.g. the query –

select *name, instructor_r.dept_name, building*
from *instructor_r, department_r*
where instructor_r.dept_name = department_r.dept_name;

The output of this query will be –

name	dept_name	building
George	Computer	Building No. 1
Krishna	Computer	Building No. 1
Uday	IT	Building No. 2

We can give other query to get the **same output as above** –

select *name, I.dept_name, building*
from *instructor_r* ***as*** *I, department_r as D*
where I.dept_name = D.dept_name;

In the above query, I and D are declared as aliases, that is alternative names, for the relation instructor_r and department_r. An identifier, such as I and D, that is used to rename a relation is referred to as a **correlation name** in the SQL standard, but is also commonly referred to as a **table alias**, or a **correlation variable**, or a **tuple variable**.

✓ String Operations

- SQL specifies strings by enclosing them in single quotes, for example, 'Computer'. Some systems also allow double quotes.
- Some database systems distinguish the uppercase and lowercase when matching the strings. But, some database systems such as MySQL and SQL Server do not distinguish between uppercase and lowercase.
- SQL also permits a variety of functions on character strings, such as
  - Concatenating (using "_")
  - extracting substrings,
  - finding the length of strings,
  - converting strings to uppercase using the function **upper**(*s*) where *s* is a string.
  - converting to lowercase using the function **lower**(*s*),
  - removing spaces at the end of the string using **trim**(*s*) and so on.

- Pattern matching can be performed on strings, using the operator **like**. We describe patterns by using two special characters:
  -Percent (%): The % character matches any substring.
  -Underscore (_): The character matches any character
- To illustrate pattern matching, consider the following examples:
  - 'Intro%' matches any string beginning with "Intro".
  - '%Comp%' matches any string containing "Comp" as a substring, for example, 'Intro. to Computer Science', and 'Computational Biology'.
  - '_ _ _' matches any string of exactly three characters.
  - '_ _ _ %' matches any string of at least three characters.
- E.g. consider the following query –

  select name **from** instructor_r
  where dept_name **like** 'Com%';

The output of this query will be as follow -

name
George
Krishna

Consider other query as follow -

select name, dept_name **from** instructor_r
where dept_name **like** "com%";

The output of this query will be as follow -

name	**dept_name**
George	Computer
Krishna	Computer

- SQL allows to search for mismatches instead of matches, by using the **not like** comparison operator.

E.g. consider the query as follow –

select name, dept_name **from** instructor_r
where dept_name **not like** “com%”;

The output of this query will be as follow -

name	**dept_name**
Uday	IT

✓ Attribute Specification in select clause

- The asterisk symbol “ * ” can be used to denote all attributes.
- E.g. consider the query as –

select * **from** instructor_r;

The output of this query will be as follow –

ID	**name**	**dept_name**	**salary**
100	George	Computer	110000
101	Krishna	Computer	95000
102	Uday	IT	105000

- Consider other query as follow –

select instructor_r.*
from instructor_r, department_r
where instructor_r.dept_name = department_r.dept_name
and department_r.building = ‘Building No. 1’;

The output of this query will be as follow –

ID	**name**	**dept_name**	**salary**
100	George	Computer	110000
101	Krishna	Computer	95000

- Consider other query as follow –

```
select instructor_r.*, department_r.*
from instructor_r, department_r
 where instructor_r.dept_name = department_r.dept_name
        and department_r.building = 'Building No. 1';
```

OR

```
select *
from instructor_r, department_r
 where instructor_r.dept_name = department_r.dept_name
        and department_r.building = 'Building No. 1';
```

Then output of both query will be same and as follow –

ID	name	dept_name	salary	dept_name	building	budget
100	George	Computer	110000	Computer	Building No. 1	1000000
101	Krishna	Computer	95000	Computer	Building No. 1	1000000

✓ Ordering the Display of Tuples

- The **order by** clause causes the tuples in the result of a query to appear in sorted order.
- By default, the **order by** clause lists the items in ascending order.
- To specify the sort order, we must specify **desc** for descending order or **asc** for ascending order.
- E.g. consider the query as follow –

```
select * from instructor_r order by salary;
```

OR

```
select * from instructor_r order by salary asc;
```

The output will be as follow –

ID	name	dept_name	salary
101	Krishna	Computer	95000
102	Uday	IT	105000
100	George	Computer	110000

- Consider other query as follow –

  select * **from** instructor_r **order by** salary **desc**;

  The output will be as follow –

ID	name	dept_name	salary
100	George	Computer	110000
102	Uday	IT	105000
101	Krishna	Computer	95000

✓ Where Clause Predicates

- SQL includes a **between** comparison operator to simplify **where** clauses that specify that a value be less than or equal to some value and greater than or equal to some other value.
- It means – **between** operator is used to find out all tuples whose attribute value is ranging between given range of values. It also includes the boundary values.
- E.g. consider the query

```
select name, dept_name
from instructor_r
where salary <=110000 and salary >=105000;
```

OR

```
select name, dept_name
from instructor_r
where salary between 105000 and 110000;
```

The output will be as follow -

name	dept_name
George	Computer
Uday	IT

- Similarly we can use **not between** comparison operator.
  E.g. -

```
select name, dept_name
from instructor_r
where salary not between 105000 and 110000;
```

The output will be as follow –

name	dept_name
Krishna	Computer

➢ Set Operations

- The operations **union**, **intersect**, and **except** operate on relations.
- The **union**, **intersect**, and **except**: - correspond to mathematical set-theory operations **U**, **Ո**, and **─**.
- Assume we want to perform those operations on two queries – then both the queries must be "**union compatible**". It means that both return the same number of columns, and that the corresponding columns have same (compatible) data types.

✓ **The Union operation – say R1 U R2**, then the result contains all tuples (rows) from relation R1 and all rows from relation R2 and duplicate rows are deleted. i.e. duplicate rows are put only once the resultant relation.

- Consider the table of Figure 3.17:- instructor_r relation.

ID	name	dept_name	salary
100	George	Computer	110000
101	Krishna	Computer	95000
102	Uday	IT	105000

and other table instructor_nba as follow:-

ID	name	dept_name	salary
101	Krishna	Computer	95000
102	Uday	IT	105000
103	Swapnil	ETC	112000
104	Soumitra	Mechanical	115000

Figure 3.20:- instructor_nba relation

- Now if the query fired is as follow –

```
select name from instructor_r
union
select name from instructor_nba;
```

Then the resultant relation will be as follow:

name
George
Krishna
Uday
Swapnil
Soumitra

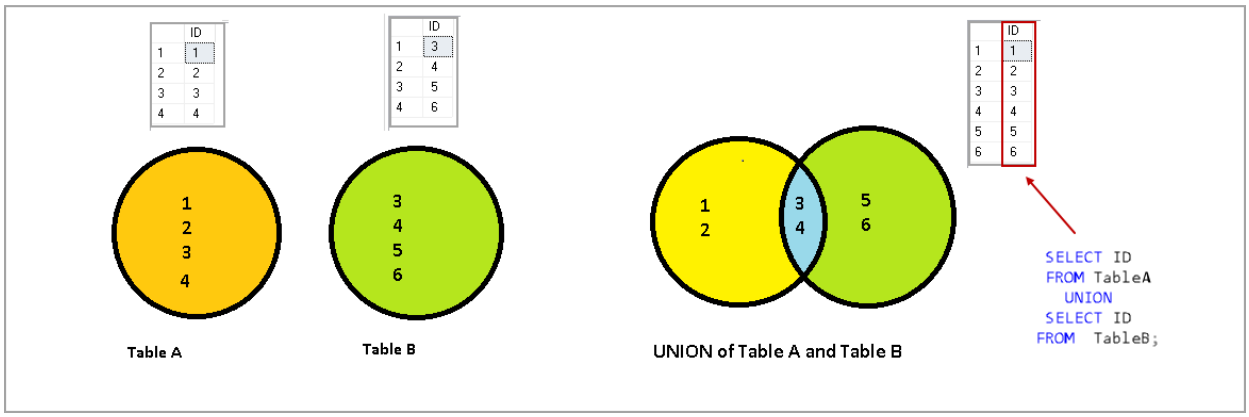
- If we want to retain all duplicates, then we must write **union all**. E.g.

```
select name from instructor_r
union all
select name from instructor_nba;
```

Then the resultant relation will be as follow:

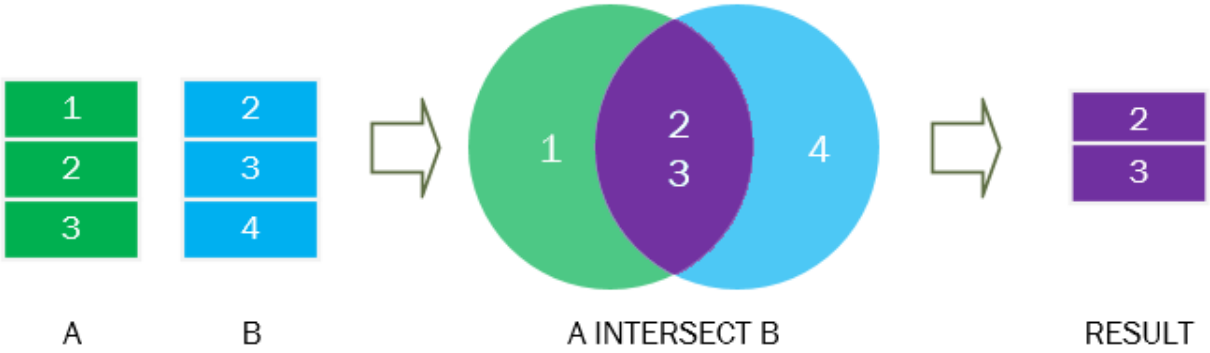
name
George
Krishna
Uday
Krishna
Uday
Swapnil
Soumitra

- The union operation is explained with figure -



✓ The Intersect Operation

- The **R1 ∩ R2** operation – It gives the values which are common in both the tables R1 and R2.
- The intersect operation automatically eliminates the duplicates. Consider the following example.
- The intersect operation can be explained as –



- The intersection operation for the tables instructor_r and instructor_nba will be as follow –
  E.g. the query is :

```
select name from instructor_r
intersect
select name from instructor_nba;
```

  Note: - MySQL (or some version) doesn't support intersect operator.

  So the equivalent query in MySQL will be as follow:-

```
select instructor_r.name
from instructor_r
join instructor_nba
on instructor_r.name = instructor_nba.name;
```

  So the output will be

name
Krishna
Uday

- If the query is as follow –

```
select instructor_r.*
from instructor_r
join instructor_nba
on instructor_r.name = instructor_nba.name;
```

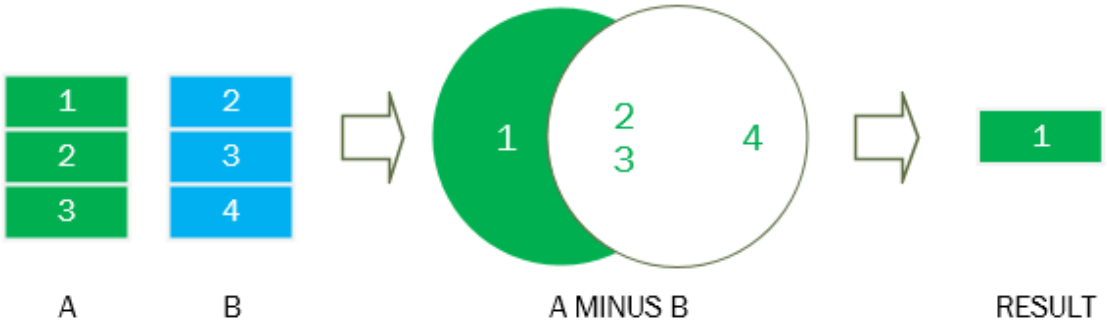
The output will be as follow -

ID	name	dept_name	salary
101	Krishna	Computer	95000
102	Uday	IT	105000

- If we want to retain all duplicates, we must write **intersect all** in place of **intersect**.

✓ The Except Operation

- The **except** operation outputs all tuples from its first input that do not occur in the second input; that is, it performs set difference. The operation automatically eliminates duplicates in the inputs before performing **set difference**.
- To retain duplicates, use **except all** in place of **except**.
- Consider the following example :



- Note: - MySQL (or some version) doesn't support **except** operator.
- Consider the following query in any other DBMS –

```
select * from instructor_r
except
select * from instructor_nba;
```

Then the resultant relation will be as follow -

ID	name	dept_name	salary
100	George	Computer	110000

➢ Null Values

- The null value is a special character value that signifies that the value is **unknown** or **does not exist**.
- Null values present special problems in relational operations, including arithmetic operations, comparison operations, and set operations.
- The result of an arithmetic expression (involving, for example +, −, *, or /) is null, if any of the input values is null.
- Sometimes null values cannot be avoided.
  E.g. consider the following relational schema:-
  STUDENT-PHONE

Stu-ID	S-Phone

  Now here, the student may not have any phone number. So, he/she will put a null value.
- Generally, database does not allow the primary key to contain null values.
- Since the predicate in a **where** clause can involve Boolean operations such as **and**, **or**, and **not** on the results of comparisons, the definitions of the Boolean operations are extended to deal with the value **unknown**.
  - **and**: The result of *true* **and** *unknown* is *unknown*, *false* **and** *unknown* is *false*, while *unknown* **and** *unknown* is *unknown*.
  - **or**: The result of *true* **or** *unknown* is *true*, *false* **or** *unknown* is *unknown*, while *unknown* **or** *unknown* is *unknown*.
  - **not**: The result of **not** *unknown* is *unknown*.
- SQL uses the special keyword **null** in a predicate to test for a null value.
  E.g. consider the following query –

```
select name
from instructor_r
where salary is null;
```

The output will be empty set.

Now consider the query as follow:-

select name
from instructor_r
where salary **is not null**;

The result of this query will be as follow:-

name
George
Krishna
Uday

➢ Aggregate Functions

- **Aggregate functions** are functions that take a collection of values as input, and return a single value. SQL offers five built-in aggregate functions:
  - Average: **avg**
  - Minimum: **min**
  - Maximum: **max**
  - Total: **sum**
  - Count: **count**

✓ Basic Aggregation

❖ avg ( ) :-

- This **avg** ( ) function is used to calculate average of column values in a table selected by the query.
- It returns the average of all the values in the specified column.
- E.g. consider the query – find out the average salary of instructor from the table instructor_r.

select avg (salary)
from instructor_r;

The resultant relation will be –

avg (salary)
103333. 3333333

- Consider the query – find out the average salary of instructor of computer department from the table instructor_r.

  select avg (salary)
  from instructor_r
  where dept_name = 'Computer';

  The resultant relation will be –

avg (salary)
102500. 0000000

  Here avg (salary) → any arbitrary name given to the attribute of resultant relation

- Consider the query as follow–

  select avg (salary) **as** Average_Salary
  from instructor_r
  where dept_name = 'Computer';

  The resultant relation will be –

Average_Salary
102500. 0000000

❖ count ( ) :-

- This function is used to calculate number of rows (tuples or records) in a table selected by the query.
- Consider the query – find out the total number of instructors in the table instructor.

  select count (ID) **as** Total_Instructor
  from instructor;

  OR

  select count (name) **as** Total_Instructor
  from instructor;

OR

```
select count (*) as Total_Instructor
from instructor;
```

The result will be as follow –

Total_Instructor
11

- To eliminate the duplicates, use the keyword **distinct**.
- E.g. query is – find out total number of departments from instructor table-

```
select count (distinct dept_name) as Total_Department
from instructor;
```

The result will be as follow **-**

Total_Department
05

❖ sum ( ) :-

- This function is used to calculate the sum of column values in a table selected by the query.
- E.g. consider the query – find out total salary paid to all instructors, from instructor table.

```
select sum(salary) as Total_Salary
from instructor;
```

The output will be as follow **–**

Total_Salary
1112000.000

❖ min ( ) :-

- This function is used to find minimum value out of a column values in a table selected by the query.

- E.g. consider the query – find out minimum salary paid to the instructors, from instructor table.

```
select min(salary) as Min_Salary
from instructor;
```

The output will be as follow –

Min_Salary
70000.000

❖ max ( ) :-

- This function is used to find maximum value out of a column values in a table selected by the query.
- E.g. consider the query – find out maximum salary paid to the instructors, from instructor table.

```
select max(salary) as Max_Salary
from instructor;
```

The output will be as follow –

Max_Salary
115000.000

✓ Aggregation with Grouping

- Related rows can be grouped together by **Group By** clause based on distinct values that exist for specified columns.
- The **Group By** clause when used in a select command, divides the relation into groups on the basis of values of one or more attributes. After dividing the relation into groups, the aggregate functions can be applied on the individual group independently. The aggregate functions are performed separately for each group and return the corresponding result value separately.
- E.g. consider the query – find out the average salary of each department, from instructor table:-

```
select dept_name, avg(salary) as Average_Salary
from instructor
group by dept_name;
```

The output will be as follow –

dept_name	Average_Salary
Computer	102500.0000000
IT	87500.0000000
ETC	102333.3333333
Mechanical	108000.0000000
Electrical	104500.0000000

- Consider other query – find out total number of instructors in each department, from instructor table.

```
select dept_name, count(dept_name) as Total_Instructors_in_Dept
from instructor
group by dept_name;
```

The output will be as follow –

dept_name	Total_Instructors_in_Dept
Computer	2
IT	2
ETC	3
Mechanical	2
Electrical	2

- Consider other query – find out total number of instructors teaching building wise-

```
select building , count(dept_name) as Total_Instructor_in_Building
from instructor natural join department
group by building;
```

The output will be as follow –

building	Total_Instructor_in_Building
Building No. 1	5
Building No. 2	4
Building No. 3	2

✓ The Having Clause

- Many times it is useful to state a condition that applies to groups rather than to tuples.
- The **Having** clause is like a **Where** clause, but applicable only to groups as a whole, whereas the where clause applies to the individual rows.
- In short – The having clause is used to filter the rows after grouping them.
- E.g. we are interested in only those departments where the average salary of the instructor is more than 100000.

```
select dept_name, avg(salary) as Average_Salary
from instructor
group by dept_name
having avg(salary) >100000;
```

The resultant relation will be as follow –

dept_name	Average_Salary
Computer	102500.0000000
ETC	102333.3333333
Mechanical	108000.0000000
Electrical	104500.0000000

- In general, the order of clauses in **select** statement will be as follow-

```
select      column(s)
from        tables(s)
where       row condition(s)
group by    columns(s)
having      group of rows condition(s)
```

- E.g. consider other query – find out total number of instructors teaching building wise, but don't consider the instructors of those departments whose budget is less than 500000. Then display the result only if total number of instructors teaching in that building is greater than 3.

  select building , **count**(dept_name) **as** Total_Instructor_in_Building
  from instructor **natural join** department
  where budget >500000
  group by building
  having count*(dept_name) >3*;

  The result of this query will be as follow -

building	**Total_Instructor_in_Building**
Building No. 1	5

➢ Nested Subqueries

- A query within another query is called as **subquery** (or **nested query**).
- It is normally a **select** statement inside **select** statement. It can be used with **INSERT, UPDATE** and **DELETE** statements
- The subquery (i.e. **inner query**) executes before the main query. The result of the inner query is used by the main query (i.e. **outer query**).
- A common use of subqueries is to perform tests for set membership, make set comparisons, and determine set cardinality, by nesting subqueries in the **where** clause.
- General Syntax :-

```
select      select_list
from        table
where       Expr_Operator
            (select     select_list
            from        table);
```

- The various operators that can be used are :- <, >, <=, >=, < >, IN, ANY, ALL, EXISTS etc.
- E.g. consider the query as – find out the instructors whose salary is less than Uday's salary.

```
select name, salary
from instructor
where salary <
        (select salary
         from instructor
         where name = 'Uday');
```

The output of this query will be as follow;-

name	**salary**
Krishna	95000.000
Manas	98000.000
Navin	101000.000
Devendra	97000.000
Parth	103000.000
Nilesh	70000.000

✓ Set Membership

- SQL allows testing tuples for membership in a relation.
- The **IN** operator is used for set membership. The set is a collection of values produced by a select clause.
- The meaning of **IN** operator – Equal to any member in the list.
- The **NOT IN** operator tests for the absence of set membership.
- E.g. consider the query – find out the names of instructors (along with their ID, their department and salary), who belong to Computer or IT department.

  The query can be given as follow –

```
select *
from instructor
where dept_name = 'computer' or dept_name ='IT';
```

Now other query can also be fired to get same output, using **in** operator.

```
select *
from instructor
where dept_name in ('Computer', 'IT');
```

The output for both the queries will be same and follows-

ID	name	dept_name	salary
100	George	Computer	110000.000
101	Krishna	Computer	95000.000
102	Uday	IT	105000.000
110	Nilesh	IT	70000.000

- This example shows that it is possible to write the same query several ways in SQL. This flexibility is beneficial, since it allows a user to think about the query in the way that seems most natural.
- Consider the query -- find out the names of instructors (along with their ID, their department and salary), who don't belong to Computer or IT department.

```
select *
from instructor
where dept_name not in ('Computer', 'IT');
```

The output for the above query will be as follows-

ID	name	dept_name	salary
103	Swapnil	ETC	112000.000
104	Soumitra	Mechanical	115000.000
105	Manas	ETC	98000.000
106	Pawan	Electrical	106000.000
107	Navin	Mechanical	101000.000
108	Devendra	ETC	97000.000
109	Parth	Electrical	103000.000

- Consider the query – find out names of instructors and their departments, whose departmental budget is more than 700000.

```
select name, dept_name
from instructor
where dept_name in
                    (select dept_name
                     from department
                    where budget >700000);
```

The output of this query will be as follow –

name	**dept_name**
George	Computer
Krishna	Computer
Swapnil	ETC
Soumitra	Mechanical
Manas	ETC
Navin	Mechanical
Devendra	ETC

✓ Set Comparison

- The **ANY** operator (or **SOME** operator) compares a value with any of the values in a list or returned by the subquery. This operator returns a false value if the subquery returns no tuple.
- E.g. The phrase "greater than at least one" is represented by **>some** or **>any**.
- Consider the query – find out the names of all instructors whose salary is greater than at least one instructor in the Electrical department.

```
select name from instructor
where salary >some
                    (select salary
                     from instructor
                     where dept_name='Electrical');
```

The output of this query will be as follow-

name
George
Uday
Swapnil
Soumitra
Pawan

- The > **some** comparison in the **where** clause of the outer **select** is true, if the *salary* of the tuple is greater than at least one member of the set of all salary values for instructors in Electrical department.
- SQL also allows < **some**, <= **some**, >= **some**, = **some**, and <> **some** comparisons.
- Now consider the query – find out the names of all instructors that have a salary greater than that of each instructor in the Electrical department. So we have to use "greater than all" construct. It is represented by **>all.**

```
select name from instructor
where salary >all
                    (select salary
                     from instructor
                     where dept_name='Electrical');
```

The output of this query will be as follow –

name
George
Swapnil
Soumitra

- SQL also allows < **all**, <= **all**, >= **all**, = **all**, and <> **all** comparisons.

✓ Test for Empty Relations

- SQL includes a feature for testing whether a subquery has any tuples in its result.
- The EXISTS operator evaluates to true if a subquery returns at least one tuple as a result; otherwise, it returns a false value.
- Consider the Figure 3.2:- The course relation as follow -

course_id	title	dept_name	credits
ETC-101	Analog Electronics	ETC	3
ETC-102	Microproceesor	ETC	4
ETC-103	Advanced Microprocessor	ETC	3
Mech-101	Engineering Drawing	Mechanical	3
Mech-102	Machine Design	Mechanical	4
Comp-101	Compiler Design	Computer	4
Comp-102	DBMS	Computer	4
Comp-103	Data Mining	Computer	4
Elect-101	Power Electronics	Electrical	3
Elect-102	Electromagnetic Field	Electrical	4
IT-101	Computer Network	IT	4
IT-102	Network Security	IT	3

Figure: The *course* relation

and also consider the Figure 3.3:- The prereq relation as follow-

course_id	prereq_id
IT-102	IT-101
Mech-102	Mech-101
Comp-103	Comp-102
ETC-103	ETC-102

Figure: The *prereq* relation

- Now consider the query as – find out the title of subject (course) and the corresponding department name, if and only if the subject is having any prerequisite subject.

The query can be written as follow –

```
select title, dept_name
from course
where exists
        (select *
        from prereq
        where course.course_id=prereq.course_id);
```

The output of this query will be as follows-

title	dept_name
Data Mining	Computer
Advanced Microprocessor	ETC
Network Security	IT
Machine Design	Mechanical

- Now consider the query as – find out the title of subject (course) and the corresponding department name, if and only if the subject belongs to Electrical department and is having any prerequisite subject.
  The query can be written as follow –

```
select title, dept_name
from course
where exists
        (select *
        from prereq
        where course.course_id= 'Elect%');
```

  The result of this query is empty relation.
- We can test for the nonexistence of tuples in a subquery by using the **not exists** construct.
- **Correlated Subquery:-** A correlated subquery is a subquery which is executed once for each candidate row considered by the main query. It uses a value from a column in the outer query. It causes the correlated subquery to be processed in a different way from the ordinary nested subquery. In normal subquery, the inner select runs first and executes

once. It returns values to main query. The correlated subquery executes once for each row considered by the outer query.

✓ Test for the Absence of Duplicate Tuples

- SQL includes a Boolean function for testing whether a subquery has duplicate tuples in its result.
- The **unique** construct returns the value true if the argument subquery contains no duplicate tuples.
- This unique construct is not yet widely implemented.
- We can test for the existence of duplicate tuples in a subquery by using the **not unique** construct.
- Formally, the unique test on a relation is defined to fail if and only if the relation contains two tuples t1 and t2 such that t1=t2.

✓ Subqueries in the from clause

- SQL allows a subquery expression to be used in the **from** clause.
- E.g. we have already considered the query – find out only those departments where the average salary of the instructor is more than 100000. It was written using having clause as follow-

  select dept_name, **avg**(salary) **as** Average_Salary
  from instructor
  group by dept_name
  having avg(salary) >100000;
- Now we can rewrite this query, without using having clause, but by using a subquery in the **from** clause –

  select dept_name, Average_Salary
  from (**select** dept_name, avg(salary) as Average_Salary
  from instructor
  group by dept_name)
  where Average_Salary > 100000;

  But this query does not generate the answer and gives error information as → Error code 1248: Every derived table must have its own alias.

- So, we can give the name to a relation resulted from the subquery, and rename the attributes, using the **as** clause as shown below-

```
select dept_name, Average_Salary
from (select dept_name, avg(salary) as Average_Salary
      from instructor
      group by dept_name)
      as dept_avg (dept_name, Average_Salary)
      where Average_Salary > 100000;
```

The resultant relation will be as follow -

dept_name	**Average_Salary**
Computer	102500.0000000
ETC	102333.3333333
Mechanical	108000.0000000
Electrical	104500.0000000

✓ The with clause

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the with clause occurs.
- i.e. In MySQL every query generates a temporary result or relation. In order to give a name to those temporary result set, a **with** clause is used.

✓ Scalar Subqueries

- A Scalar subquery is a subquery that returns exactly one column value per row.
- E.g. a subquery can be used in the **select** clause as shown in the following example that lists all departments along with the number of instructors in each department:-

```
select dept_name,
        (select count(*)
        from instructor
        where department.dept_name = instructor.dept_name)
        as Num_Of_Instructors
        from department;
```

The output of above query will be as follow:-

dept_name	Num_Of_Instructors
Computer	2
Electrical	2
ETC	3
IT	2
Mechanical	2

- The above example also illustrates the usage of correlation variables, that is, attributes of relations in the **from** clause of the outer query, such as *department.dept name* in the above example.
- Scalar subqueries can occur in **select**, **where**, and **having** clauses.

➢ Modification of the Database

We have to see how to add, remove or change the information with SQL.

✓ Deletion

- We can delete only complete tuples.
- We cannot delete values on only any particular attributes.
- The syntax for delete will be as follow:-

```
delete from r
where P;
```

  where, P→ predicate and r → relation.
- E.g. consider the query – delete a record (tuple) of Madan from the table student.

```
delete from student
where fname = 'Madan';
```

We can check it by using following query –

select * **from** student;

If we give the following query –

delete from student;

then all tuples from student will be deleted. Here the student relation itself still exists, but it will be empty.

- E.g. consider other query – delete all instructors having salary between 100000 to 125000.

```
delete from instructor
where salary between 100000 and 125000
```

- E.g. consider other query – delete all records from the instructor relation, for those instructors associated with a department located in building No. 1.

```
delete from instructor
where dept_name in
                    ( select dept_name
                      from department
                      where building = 'building No. 1');
```

✓ Insertion

- We have to see how to insert data into a relation.
- The simplest insert statement is a request to insert data into one tuple.
- The attribute values for inserted tuples must be members of the corresponding attribute's domain. Similarly, tuples inserted must have the correct number of attributes.
- E.g. we have created a table instructor with following SQL command:-

```
create table instructor
                    (ID int,
                    name varchar(25) not null,
                    dept_name varchar(15) not null,
                    salary numeric (10,3),
                    primary key (ID));
```

- Now check it using following commands-

  show tables;

  desc instructor;
- Now the records can be inserted using insert into command :

  insert into instructor

  values (100,'George', 'Computer', 110000);

  To insert second record (tuple) -

  insert into instructor

  values (101,'Krishna', 'Computer', 95000);

- In above both examples, the values are specified in the order in which the corresponding attributes are listed in the relation schema.
- If you don't remember the order of attributes, the attributes can be specified in insert statement as follow –

  insert into instructor (ID, name, dept_name, salary)

  values (100,'George', 'Computer', 110000);

  insert into instructor (name, ID, salary, dept_name)

  values ('Krishna', 101, 95000, 'Computer');
- It is possible to give only some values of attributes, while remaining attributes can be assigned NULL values (provided that NOT NULL constraint is not specified for that attribute). E.g.

  insert into instructor (ID, name, dept_name, salary)

  values (111,'Vishal', 'Mechanical', null);

✓ Updates

- The **UPDATE** command is used to change the value of an attribute in the existing records of a table.
- E.g. consider a table *labs*:-
  Give the following MySQL command;

  desc labs;

Let the output is as follows –

Field	Type	Null	Key	Default	Extra
labno	int	NO	PRI	NULL	
labname	varchar(15)	YES		NULL	
NoOfcomp	int	YES		NULL	

select * from labs;

Let the output is as follows –

labno	labname	NoOfcomp
1	PC-LAB	20
2	PC-AT-LAB	21
3	H/w lab	NULL

Now if the query -- we want to change the number of computers in labno 1 to 25, then following command will be used:

update labs
set NoOfComp= 25
where labno=1;

We can check it by using select command as follows –

select * from labs;

The output will be as follows –

labno	labname	NoOfcomp
1	PC-LAB	25
2	PC-AT-LAB	21
3	H/w lab	NULL

- E.g. consider table faculty. Now give the following command to see the details of existing records in that table –

select * from faculty;

Let the output is as follow -

fid	fname	fsalary	fphone
1	Mukul	90000	NULL
2	Rohit	100000	NULL
3	Ashok	90000	123456789
4	Baban	65000	NULL

Now if the query is – increase the salary of faculty by 1000. Then the following command will be used –

update faculty
set fsalary=fsalary+1000;

Now check it by using following command;

select * from faculty;

The output will be as follow -

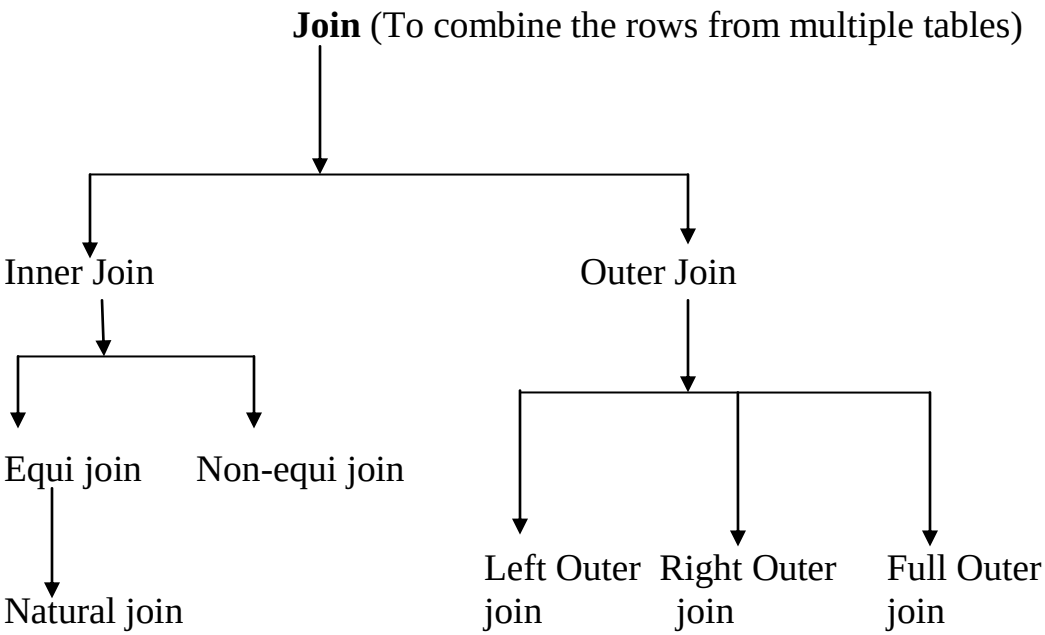
fid	fname	fsalary	fphone
1	Mukul	91000	NULL
2	Rohit	101000	NULL
3	Ashok	91000	123456789
4	Baban	66000	NULL

Unit-III

(Intermediate SQL)

➢ Join Expressions

- Refer **natural join**, page number 28.
- SQL provides other forms of the join operation.
- **In General**, following are the types of Joins :



- Equi-join – This join condition consists only of equality condition.
- Natural Join - The SQL NATURAL JOIN is a type of EQUI JOIN and is structured in such a way that, columns with the same name of associated tables will appear once only.
- Non-equi join - Equijoin is a comparator-based join which uses equality comparisons in the join-predicate. However, if we use other comparison operators like >, <, >=, <=, < > etc. **(i.e. other**

than =) it can't be called equijoin, but it will be called as Non-equi join.

✓ Join Conditions

- The **on** condition allows a general predicate over the relations being joined. This predicate is written like a **where** clause predicate except for the use of the keyword **on** rather than **where**.
- E.g. consider the relations instructor and department in a reduced form (i.e. less number of records), named as instructor_r and department_r. **(Refer figure 3.17 and figure 3.18) –**

instructor_r relation

ID	**name**	**dept_name**	**salary**
100	George	Computer	110000
101	Krishna	Computer	95000
102	Uday	IT	105000

department_r relation

dept_name	**building**	**budget**
Computer	Building No. 1	1000000
IT	Building No. 2	500000
ETC	Building No. 1	800000

Now if the following query is issued (**Example of equi-join**) –

select *
from instructor_r, department_r
where instructor_r.dept_name=department_r.dept_name;

The resultant relation will be as follows:

ID	**name**	**dept_name**	**salary**	**dept_name**	**building**	**budget**
100	George	Computer	110000.000	Computer	Building No. 1	1000000.00
101	Krishna	Computer	95000.000	Computer	Building No. 1	1000000.00
102	Uday	IT	105000.000	IT	Building No. 2	500000.00

Now if the following query is issued (**Example of natural join**) –

```
select *
from instructor_r
natural join department_r;
```

The resultant relation will be as follows:

ID	name	dept_name	salary	building	budget
100	George	Computer	110000.000	Building No. 1	1000000.00
101	Krishna	Computer	95000.000	Building No. 1	1000000.00
102	Uday	IT	105000.000	Building No. 2	500000.00

Now if the following query is issued (**Example of join on**) –

```
select *
from instructor_r,
join department_r
on instructor_r.dept_name=department_r.dept_name;
```

OR

```
select *
from instructor_r,
inner join department_r
on instructor_r.dept_name=department_r.dept_name;
```

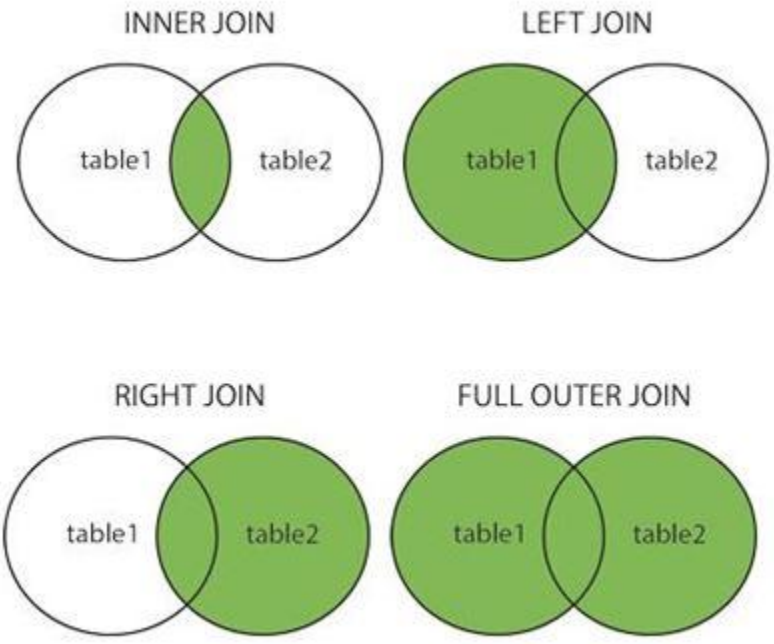
The resultant relation will be as follows:

ID	name	dept_name	salary	dept_name	building	budget
100	George	Computer	110000.000	Computer	Building No. 1	1000000.00
101	Krishna	Computer	95000.000	Computer	Building No. 1	1000000.00
102	Uday	IT	105000.000	IT	Building No. 2	500000.00

✓ Outer Joins

- In an inner join, the rows from both the tables where the join gets matched are selected. So some rows (tuples) in any of the tables may be lost. It happens because the rows that do not satisfy the join condition are rejected.

- The **outer join** operations works in a manner similar to the inner join operations, but preserve those tuples that would be lost in a join, by creating tuples in the result containing null values.
- In general, there are three forms of outer join:
    - **Left join (left outer join)**:- This join returns all rows from the left table and only those rows from right table where join condition is met.
    - **Right join (right outer join)**:- This join returns all rows from the right table and only those rows from left table where join condition is met.
    - **Full join (full outer join):-** It is a combination of the left and right outer join types. The resultant relation contains all rows from both the tables, with null values for the missing rows where join condition is not met.



- Consider two new tables for the explanation.
  department_all → a table containing two columns - d_id (i.e. department id) and corresponding dept_name.

employee → a table containing three columns – EID (employee id), Ename (employee name) and E_did (employee's department id).

Assume following is the instance of both the relations –

department_all

d_id	dept_name
1	Computer
2	IT
3	ETC
4	Mechanical

employee

EID	Ename	E_did
100	George	1
101	Krishna	1
102	Uday	2
200	Bhaskar	15

- **Left join (left outer join)**:-
  E.g. the following query is issued –

```
select *
from employee
left outer join department_all
on employee.E_did=department_all.d_id;
```

OR

```
select *
from employee
left join department_all
on employee.E_did=department_all.d_id;
```

Then the resultant relation will be as follows:

EID	Ename	E_did	d_id	dept_name
100	George	1	1	Computer
101	Krishna	1	1	Computer
102	Uday	2	2	IT
200	Bhaskar	15	NULL	NULL

Consider other example – if the following query is issued –

```
select *
from department_all
left join employee
on employee.E_did=department_all.d_id;
```

Then the resultant relation will be as follow-

d_id	dept_name	EID	Ename	E_did
1	Computer	100	George	1
1	Computer	101	Krishna	1
2	IT	102	Uday	2
3	ETC	NULL	NULL	NULL
4	Mechanical	NULL	NULL	NULL

- **Right join (right outer join)**:- The right outer join is symmetric to the left outer join. Tuples from the right-hand side relation that do not match any tuple in the left-hand side relation are padded with nulls and are added to the result of the right outer join.

  E.g. following query is issued –

```
select *
from employee
right join department_all
on employee.E_did=department_all.d_id;
```

  OR

```
select *
from employee
right outer join department_all
on employee.E_did=department_all.d_id;
```

Then the resultant relation will be as follow-

EID	Ename	E_did	d_id	dept_name
100	George	1	1	Computer
101	Krishna	1	1	Computer
102	Uday	2	2	IT
NULL	NULL	NULL	3	ETC
NULL	NULL	NULL	4	Mechanical

- **Full join (full outer join):-** The full join clause is not supported by MySQL. So the operation can be carried out by combination of left join and right join as follows-

```
select *
from employee
left join department_all
on employee.E_did=department_all.d_id
union all
    select *
    from employee
    right join department_all
    on employee.E_did=department_all.d_id;
```

Then the resultant relation will be as follow-

EID	Ename	E_did	d_id	dept_name
100	George	1	1	Computer
101	Krishna	1	1	Computer
102	Uday	2	2	IT
200	Bhaskar	15	NULL	NULL
100	George	1	1	Computer
101	Krishna	1	1	Computer
102	Uday	2	2	IT
NULL	NULL	NULL	3	ETC
NULL	NULL	NULL	4	Mechanical

➢ Views

- **Views** are generally known as **virtual tables or virtual relations**.
- When we use the CREATE TABLE command to create a table, then physically the space is created for this table and is stored in database.
- But it is not desirable for all users to see the entire logical model and all relations. Security constraints may require that certain data must be hidden from some users.
- So, SQL allows a virtual relation to be defined by a query, and the relation conceptually contains the result of the query. The virtual relation is not precomputed and stored, but instead is computed by executing the query whenever the virtual relation is used.
- **In simple words** - In Views, the resulting records of the query expression is not stored on the disk, but only the query expression is stored on the disk. So query expression gets executed every time when user tries to get the data from it, so that user will get the latest updated value every time.

✓ View Definition and using view in SQL queries

- The view in SQL is defined by **create view** command.
- To define a view, we must give -- the name to a view and the query that computes the view.
- The syntax for **create view** command is as follows-

  create view V **as** <query expression>;

  where, V → view name
- E.g. consider the tables from figure 3.2 and 3.3 as shown below-

prereq relation

course_id	**prereq_id**
IT-102	IT-101
Mech-102	Mech-101
Comp-103	Comp-102
ETC-103	ETC-102

Course relation

course_id	**title**	**dept_name**	**credits**
ETC-101	Analog Electronics	ETC	3
ETC-102	Microproceesor	ETC	4
ETC-103	Advanced Microprocessor	ETC	3
Mech-101	Engineering Drawing	Mechanical	3
Mech-102	Machine Design	Mechanical	4
Comp-101	Compiler Design	Computer	4
Comp-102	DBMS	Computer	4
Comp-103	Data Mining	Computer	4
Elect-101	Power Electronics	Electrical	3
Elect-102	Electromagnetic Field	Electrical	4
IT-101	Computer Network	IT	4
IT-102	Network Security	IT	3

- Now, if we want to find out the courses IDs and title of the subjects that have some prerequisite courses, along with department names to which they belong. So first create the view using **create view** command as follow –

```
create view course2 as
        select prereq.course_id, title, dept_name
        from course, prereq
        where course.course_id = prereq.course_id;
```

So view i.e. virtual relation will be created. Now give the following command to get the desired result –

```
select * from course2;
```

The output of this query will be as follow –

course_id	**title**	**dept_name**
Comp-103	Data Mining	Computer
ETC-103	Advanced Microprocessor	ETC
IT-102	Network Security	IT
Mech-102	Machine Design	Mechanical

- Now if the query is -- to find out the courses IDs and title of the subjects that have some prerequisite courses, in computer department. Then the query can be issued as follow –

```
select * from course2
where dept_name = 'Computer';
```

The output of this query will be as follow –

course_id	title	dept_name
Comp-103	Data Mining	Computer

✓ Materialized Views

- Certain database systems allow view relations to be stored, but they make sure that, if the actual relations used in the view definition change, the view is kept up-to-date. Such views are called as materialized views.
- The process of keeping the materialized view up-to-date is called **materialized view maintenance** (or **view maintenance**).
- View maintenance can be done immediately when any of the relations on which the view is defined is updated. Some database systems, however, perform view maintenance lazily, when the view is accessed.
- Applications that use a view frequently may benefit if the view is materialized. Applications that demand fast response to certain queries can benefit greatly by creating materialized views corresponding to the queries.
- The materialized view has the disadvantages also – e.g. storage costs and the added overhead for updates.
- MySQL does not support the materialized view by itself, but the effect of it can be generated.

✓ Update of a View

- Views present serious problems if we express updates, insertions, or deletions with them. The difficulty is that a modification to the

database expressed in terms of a view must be translated to a modification to the actual relations in the logical model of the database.

- Because of problems, modifications are generally not permitted on view relations, except in limited cases. Different database systems specify different conditions under which they permit updates on view relations.
- In general, an SQL view is said to be **updatable** (that is, inserts, updates or deletes can be applied on the view) if the following conditions are all satisfied by the query defining the view (some of the conditions have been mentioned here):-
    - It should not contain aggregate functions (e.g. **sum( ), min( ), max( )** etc.
    - The query should not have a **group by**, **having** or **distinct** clause.
    - The query should not have a **union** or **union all** clause.
    - Should not contain subquery in the select list.
    - Should not contain a subquery in **where** clause that refers to a table in **from** clause.

➢ Integrity Constraints

- Also refer the topic "Basic Schema Definition", from Page No. 17.
- The integrity constraints ensure that:- the changes made in the database are by authorized users and consistency of data in the database is maintained i.e. no loss of data consistency.
- Integrity constraints guard against accidental damage to the database.
- Examples of integrity constraints –
    - An instructor name in instructor relation, cannot be null.
    - No two instructors in instructor relation, can have the same ID, etc
- Integrity constraints are identified at –
    - database schema designing process – using **create table** command

- they can be added to existing relation – using **alter table** table-name **add** constraint.

✓ Constraints on a Single Relation

- The create table command may also include integrity constraint statements.
- The integrity constraints can be as follow:
  - **Domain Constraint**: - Domain constraints are the most elementary forms of integrity constraints. It can be specified for each attribute by defining its specific range or domain. It requires that the attribute value must fall under a particular range in order to be valid. Thus, the domain of the salary attribute of the instructor relation is the set of all possible salary values.
  - **Primary key** :- The primary key attributes are required to be nonnull and unique (i.e. no duplicate) ; that is, no tuple can have a null value or a duplicate value for a primary key attribute.
  - **Not null :-** The not null constraint on an attribute specifies that the null value is not allowed for that attribute.
    E.g. the null value may be inappropriate for instructor name in a instructor relation. This can be done at the time of creating a table by using **create table** clause, by including –
    name **varchar**(25) **not null**
    The use of not null, prevents the insertion of a null value for the attribute name.
    The primary key need not to be declared explicitly as not null.
  - **Unique Constraint:-** This constraint ensures that an attribute or set of attributes has unique (i.e. no duplicate) values.
    E.g. tname2 varchar(25) **not null unique**

    If unique constraint is to be applied on more than one attribute, then –
    E.g.

Fname **varchar**(25) **not null**,
Address **varchar**(25) **not null**,
unique (Fname, Address)

Here (Fname, Address) is a set of attributes and unique constraint is applied to the combination of attributes.

- **Check Constraint:-** This check ensures that the value inserted for an attribute must satisfy the specified condition.
  E.g. the **check** clause for ensuring that the value of attribute budget in department relation, should be greater than 1000, can be specified as (in **create table** command) –
  budget **numeric** (10,2) **check** (budget >1000)
- **Referential Integrity**: - The value of foreign key in the referencing relation must exist in the primary key attribute of the referenced relation.
  The foreign key can be specified as part of the **create table** statement by using the **foreign key** clause.
  E.g. Suppose a new relation instructor_tmp is created –

```
create table instructor_tmp
   (ID int,
   name varchar(25) not null,
   dept_name varchar(15) not null,
   salary numeric (10,3),
   primary key (ID),
   foreign key (dept_name) references department(dept_name));
```

This foreign key declaration specifies that the dept_name specified in the tuple must exist in the department relation. Without this constraint, it is possible to add a record in the instructor_tmp relation, for any nonexistent dept_name.

Requirement of this form are called **Referential-integrity constraints**, or **subset dependencies.**

Consider the creation of table as follow –

```
create table instructor_tmp1
  (ID int, name varchar(25) not null,
  dept_name varchar(15) not null,
  salary numeric (10,3),
  primary key (ID),
  foreign key (dept_name) references department(dept_name),
                on delete cascade,
                on update cascade);
```

Because of the clause **on delete cascade** associated with the foreign-key declaration, if a delete of a tuple in *department* results in this referential-integrity constraint being violated, the system does not reject the delete. Similarly, the system does not reject an update to a field referenced by the constraint if it violates the constraint. The SQL also allows foreign key clause to specify actions other than cascade. E.g. **on delete set null** or **on delete set default.**

➢ Functions and Procedures

- Developers can write their own functions and procedures, store them in the database, and then invoke them from SQL statements.
- Procedures and functions allow "business logic" to be stored in the database, and executed from SQL statements.
- The function or a procedure is a set of SQL statements that performs some particular task, like in any programming language.
- SQL allows the definition of functions, procedures, and methods. These can be defined either by the procedural component of SQL, or by an external programming language such as Java, C, or C++.
- The syntax to define the function and procedure may be different from one database system to another.
- The main difference between the function and procedure is that, the function always returns a value while the procedure may return or may not return the value.

- Stored procedures are procedures or functions that are stored and executed by the DBMS at the database server machine.

✓ Declaring and invoking SQL Functions and Procedures

- The general syntax for defining the function can be written as:

```
CREATE FUNCTION name_of_function (list of parameteres)
RETURNS return_datatype
BEGIN
        Sql_statements
RETURN return_variable;
END
```

- We can define the function, that returns the count of department name from the relation instructor as follow:

```
delimiter $$
 create function dept_count3 (dept_name varchar(15))
        returns integer
   deterministic
   begin
   declare d_count3 integer;
                select count(*) into d_count3
                from instructor
                where instructor.dept_name = dept_name;
   return d_count3;
   end $$
delimiter;
```

Now this above function can be used in a query that returns department name appearing more than 2 times in a instructor relation.

```
select distinct dept_name
        from instructor
        where dept_count3(dept_name) >2;
```

The output of this query will be as follow:-

dept_name
ETC

- Other example of function – The function Hello1 takes a parameter, performs an operation and returns the result.

```
delimiter $$
create function Hello1 (string_name varchar(15))
        returns varchar(50)
   deterministic
   return concat('Hello students, ',string_name,', !!');
   end $$
delimiter ;
```

Now suppose, the query is given as follow:

```
select hello1('how are you ?');
```

Then output will be as follow:

Hello students, how are you ?, !!

- Some SQL supports functions that can return table as a result. Such functions are called **table functions**. **(Note – MySQL does not support this feature).**
- Consider the function defined as follow, which returns a table containing all the instructors of a particular department.

```
# Functions - returning the table
create function instructors_of (dept_name varchar(15))
returns table(ID int,
                name varchar(25),
                dept_name varchar(15),
               salary numeric(10,3))
return table
        (select ID, name, dept_name, salary
                from instructor
                where instructor.dept_name = instructor_of.dept_name);
```

The function can be used in a query, to get all instructors of Computer department as follow:-

select * **from table** (instructors_of('Computer'));

- SQL also supports procedures. The general syntax for the procedures is as follow:

CREATE PROCEDURE name_of_procedure (list of parameteres)
Begin
local declarations
body of procedures;
end

Parameters and local declarations are optional and if declared must be valid SQL data types. Parameters declared may have one of the three modes, namely – IN, OUT, or INOUT.

E.g. Without passing any parameters → To create a procedure that is used to retrieve the information of ETC instructors such as ID, name, dept_name and salary.

```
# To write the Procedure
 delimiter $$
 create procedure GetInstName1()
 begin
   select ID, name, dept_name, salary
   from instructor
   where dept_name='ETC';
end $$
```

Now we can invoke the stored procedure by using call statement;

call GetInstName1();

The output will be as follow:

ID	name	dept_name	salary
103	Swapnil	ETC	112000.000
105	Manas	ETC	98000.000
108	Devendra	ETC	97000.000

- Now passing the parameters → Parameters declared may have one of the three modes, namely – IN, OUT, or INOUT.
  E.g. to define the procedure that returns the count of department name, from the relation instructor:

```
# To define Procedures
delimiter $$
create procedure dept_count_proc(in dept_name varchar(15),
                                 out d_count int)
begin
        select count(*) into d_count
        from instructor
        where instructor.dept_name = dept_name;
end $$
delimiter ;
```

  Now use the call statement to find out the required count, say for ETC department -

```
        call dept_count_proc('ETC', @d_count);
```

  Now to print the count value:

```
        select @d_count;
```

  The output will be as follow:

@d_count
3

- SQL permits **more than one procedure of the same name**, so long as the number of arguments of the procedures with the same name is different. The name, along with the number of arguments, is used to identify the procedure. SQL also permits **more than one function with the same name**, so long as the different functions with the same

name either have different numbers of arguments, or for functions with the same number of arguments, they differ in the type of at least one argument.

✓ Language Constructs for Procedures and Functions

- SQL supports constructs that give it almost all the power of a general-purpose programming language. The part of the SQL standard that deals with these constructs is called the **Persistent Storage Module (PSM)**.
- The SQL supports **while** statement and **repeat** statement.
- The while loop statement allows to repeat the block of codes repeatedly, till the condition is true. The syntax for while statement is as follow:

  while Boolean expression **do**
  sequence of statements;
  end while

  E.g. we want to print even numbers upto 30. So the query using while loop will be as follow:

```
# Using while loop ...to print even numbers upto 30
delimiter $$
CREATE PROCEDURE Even_No_Loop1()
BEGIN
DECLARE EvenNo INT;
declare EvenNoString varchar(100);
SET EvenNo = 0;
set EvenNoString = '';
WHILE EvenNo <=30 DO
set EvenNoString = concat(EvenNoString, EvenNo, ',');
SET EvenNo = EvenNo + 2;
END WHILE;
select EvenNoString;
END $$
Delimiter ;
```

Now call Even_No_Loop1 to see the result:

call Even_No_Loop1;

The output is as follow;

EvenNoString
0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,

- The repeat loop statement allows repeating of the block of codes , based on a condition. The syntax for repeat statement is as follow:

repeat

sequence of statements;

until Boolean expression

end repeat

E.g. we want to print even numbers upto 40. So the query using repeat loop will be as follow:

```
# Using repeat loop ...to print even numbers upto 40
delimiter $$
CREATE PROCEDURE Repeat_Loop()
BEGIN
DECLARE EvenNo INT;
declare EvenNoString varchar(100);
SET EvenNo = 0;
set EvenNoString = '';
repeat
set EvenNoString = concat(EvenNoString, EvenNo, ',');
SET EvenNo = EvenNo + 2;
until EvenNo >40
end repeat;
select EvenNoString;
END $$
Delimiter ;
```

Now call Repeat_Loop to see the result:

call Repeat_Loop;

The output is as follow;

EvenNoString
0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,

- If-then statement :-
  The general syntax for if-then can be given as follow:

```
if Boolean expression (some condition)
        then statements;
endif;
```

  The general syntax for if-then-else can be given as below:-

```
if Boolean expression (some condition)
        then statements;
elseif Boolean expression (some condition)
        then statements;
else
        statements;
end if;
```

- # To use Procedures with if-then-else:-
  E.g. to define the procedure that returns the count of department name from the relation instructor. Also level of count should be printed:

```
delimiter $$
create procedure count_proc_ifthenelse
        (in dept_name varchar(15),
        out d_count int,
        out inst_count_level varchar(15))

begin
select count(*) into d_count
from instructor
       where instructor.dept_name = dept_name;
       if d_count >2 then
                set inst_count_level = 'Excellent';
```

```
        elseif d_count>=2 then
                set inst_count_level = 'Good';
         else
                set inst_count_level = 'Poor';
         end if;
end $$
delimiter ;
```

Now call the procedure:
call count_proc_ifthenelse('ETC', @d_count,@inst_count_level);

Now print the department count and level:
select @d_count, @inst_count_level;

The output will be as follow:

@d_count	**@inst_count_level**
3	Excellent

Now if we call the procedure with IT department:
call count_proc_ifthenelse('IT', @d_count,@inst_count_level);
select @d_count, @inst_count_level;
The output will be as follow:

@d_count	**@inst_count_level**
2	Good

➢ Triggers

- A trigger is a statement (or type of stored procedure) that is executed automatically when some modification to the database is done. Modification means – some events like insert, update, delete etc occurs.
- Triggers are also called s event-condition-action rules or ECA rules.

- The main aim of triggers is to maintain data integrity and also one can design a trigger for recording information, which can be used for auditing purposes.
- To design a trigger, we must meet two requirements:
    - Specify when a trigger is to be executed.
    - Specify the action to be taken when the trigger executes.

✓ Need for triggers

- Triggers can be used to implement certain integrity constraints that cannot be specified using the constraint mechanism of SQL. Triggers are also useful mechanisms for alerting humans or for starting certain tasks automatically when certain conditions are met. As an example, suppose a warehouse wishes to maintain a minimum inventory of each item; when the inventory level of an item falls below the minimum level, an order can be placed automatically. On an update of the inventory level of an item, the trigger compares the current inventory level with the minimum inventory level for the item, and if the level is at or below the minimum, a new order is created.
- A trigger is commonly used to enforce error checking, security and safe backing-up of data.
- Triggers can be used to replicate the data to different files for achieving the data consistency.
- Triggers must be carefully defined in order to avoid duplication or replacement of existing built-in functions.
- It can be used to improve the performance in client/server architecture. All rules will run on server and then the results from server will be presented to the client.
- The trigger code written in one database application can be used in another application, as the trigger is stored in the database.

✓ Triggers in SQL

- We will consider how to implement triggers in MySQL. The syntax for implementing the triggers may change from one DBMS to another.

- The triggers can be- Row level trigger or statement level trigger.
    - Row level trigger – An event is triggered for each row updated, inserted or deleted.
    - Statement level trigger – An event is triggered for each SQL statement executed.
- There are six types of triggers that can be made:
    - Before insert
    - After insert
    - Before update
    - After update
    - Before delete
    - After delete
- **After/Before Insert Trigger:**
  The general syntax will be as follow:

  cretate trigger trigger_name
  after/before
  insert/delete/update [OF column_name**]**
  on table_name **[**use of **OLD/NEW** clause]
  for each row
  begin
  statements
  end
- OLD and NEW qualifiers: - These qualifiers are related to row triggers and represent old and new values of a column before and after the operation. The functionality of the prefix qualifiers OLD and NEW are given in following table-

DML statement	**OLD value**	**NEW value**
DELETE	value before deleted	NULL
INSERT	NULL	Value inserted
UPDATE	value before update	Value after update

- E.g. consider we have to create a trigger on the table staff_salary, such that the net_salary is calculated by deducting 10% of TDS from salary. For that create the table, named as staff_salary.

```
# Triggers…
create table staff_salary
        (ID int,
        name varchar(25) not null,
        salary numeric (10,3),
        net_salary numeric (10,3),
        primary key (ID));

#now to create trigger ---before insert
delimiter $$
create trigger trigger_insert_salary before insert
on staff_salary for each row
begin
        set new.net_salary = new.salary - ( new.salary * 0.10);
end $$
delimiter ;

# Now insert one record
insert into staff_salary (ID, name, salary, net_salary) values (100,
'George', 110000, -10);

# Give following command
select * from staff_salary;
```

The output will be as follow:-

ID	name	Salary	net_salary
100	George	110000.000	99000.000

```
#Now insert one more record
insert into staff_salary values (101, 'Krishna', 95000, null);

# Give following command
select * from staff_salary;
```

The output will be as follow:-

ID	name	Salary	net_salary
100	George	110000.000	99000.000
101	Krishna	95000.000	85500.000

Bibliography

1. Abraham Silberschatz, Henry F. Korth, S. Sudarshan, “Database System Concepts”, 6th Edition, McGraw-Hill Hill Education.
2. Abraham Silberschatz, Henry F. Korth, S. Sudarshan, “Database System Concepts”, 4th Edition, McGraw-Hill Hill Education.
3. Ramez Elmasri and Shamkant B. Navathe “Fundamentals of Database Systems”, 5th Edition, Pearson.
4. Express Learning, “Database Management Systems”, ITL Education Solutions Limited.
5. Archana Verma, “Database Management Systems”, GenNext Publication.
6. Dr. Rajiv Chopra, “Database Management Systems (DBMS) – A Practical Approach”, 5th Edition, S. Chand Technical
7. Tanmay Kasbe, “Database Management System Concepts – A Practical Approach”, First Edition, Educreation Publishing.
8. Mahesh Mali, “Database Management Systems”, Edition 2019, TechKnowledge Publications.
9. Rajendra Prasad Mahapatra, Govind Verma, “Database Management System”, Khanna Publishing.
10. Malay K. Pakhira, “Database Management System”, Eastern Economy Edition, PHI.
11. Sarika Gupta, Gaurav Gupta, “Database Management System”, Khanna Book Publishing Edition.
12. Riktesh Srivastava, Rajita Srivastava, “Relational Database Management System”, New Age International Publishers.
13. Peter Rob, Carlos Coronel, “Database System Concepts’, Cenage Learning, India Edition
14. Bipin C. Desai, “An Introduction to Database Systems”, Galgotia Publications.
15. G.K. Gupta, “Database Management Systems”, McGraw Hill Education.
16. Shio Kumar Singh, “Database Systems – Concepts, Design and Applications”, 2nd Edition, PEARSON.
17. S.D.Joshi, “Database Management System”, Tech-Max Publication.
18. R. Ramkrishnan , J. Gehrke, "Database Management Systems", 3rd Edition, McGraw-Hill
19. C. J. Date, “Introduction to Database Management Systems”, 8th Edition, Pearson
20. Atul Kahate, “Introduction to Database Management System”, 3rd Edition, Pearson.
21. Bharat Lohiya, “Database Systems”, Tenth Edition, Aditya Publication, Amravati.
22. Vijay Krishna Pallaw, “Database Management System”, 2nd, Asian Books Pvt. Ltd.
23. Database Management Systems, Database Management Systems.
24. Mrs. Jyoti G. Mante (Khurpade), Mrs. Smita M. Dandge, “Database Mangement System”, Nirali Prakashan.
25. Step by Step Database Systems (DBMS), Shiv Krupa Publications, Akola

26. Mrs. Sheetal Gujar –Takale, Mr. Sahil K. Shah, "Database Management System", Nirali Prakashan.
27. Mrs. Jyoti G. Mante (Khurpade), U.S. Shirshetti, M.V. Salvi, K.S. Sakure, "Relational Database Management System", Nirali Prakashan.
28. Seema Kedar, Rakesh Shirsath, "Database Management Systems", Technical Publications.
29. Pankaj B. Brahmankar, "Database Management Systems", Tech-Max Publications, Pune.
30. Imran Saeed, Tasleem Mustafa, Tariq Mahmood, Ahsan Raza Sattar, "A Fundamental Study of Database Management Systems", 3rd Edition, IT Series Publication.
31. Database Management Systems Lecture Notes, Malla Reddy College of Engineering and Technology, Secunderabad.
32. Dr. Satinder Bal Gupta, Aditya Mittal, "Introduction to Database Management System, University Science Press.
33. E-Notes BCS 41/ BCA 41 on "Database Management System", Thiruvalluvar University.
34. Bighnaraj Naik, Digital Notes on "Relational Database Management System", VSSUT, Burla.
35. Viren Sir, Relational database Management System", Adarsh Institute of Technolgoyt (Poly), VITA.
36. Sitansu S. Mitra, "Principles of Relational Database Systems", Prentice Hall.
37. Neeraj Sharma, Liviu Perniu, Raul F. Chong, Abhishek Iyer, Chaitali Nandan, Adi-Cristina Mitea, Mallarswami Nonvinkere, Mirela Danubianu, "Database Fundamentals", First Edition, DB2 On Campus Book Series.
38. Database Management System, Vidyavahini First Grade College, Tumkur.
39. Bhavna Sangamnerkar, Revised by: Shiv Kishor Sharma, "Database Management System", Think Tanks Biyani Group of Colleges.
40. Tibor Radvanyi, "Database Management Systems".
41. Ramon A. Mata-Toledo, Pauline K. Cushman, "Fundamentals of Relational Databases", Schaum's Outlies.

Bibliography

Web Resources

https://www.sqlshack.com/sql-union-vs-union-all-in-sql-server/
https://www.sqltutorial.org/sql-intersect/
https://gokhanatil.com/2010/10/minus-and-intersect-in-mysql.html
https://www.sqltutorial.org/sql-minus/
https://www.w3schools.com/sql/sql_in.asp
https://www.geeksforgeeks.org/
https://www.w3resource.com/sql/joins/natural-join.php
https://www.guru99.com/joins-sql-left-right.html
https://www.w3schools.com/sql/sql_join.asp
https://www.guru99.com/joins-sql-left-right.html
http://www.sql-join.com/
https://www.geeksforgeeks.org/sql-join-set-1-inner-left-right-and-full-joins/
https://www.tutorialspoint.com/sql/sql-using-joins.htm
https://www.tutorialspoint.com/difference-between-views-and-materialized-views-in-sql
https://tutorialink.com/dbms/procedures-and-functions.dbms
https://docs.oracle.com/cd/E19078-01/mysql/mysql-refman-5.0/stored-programs-views.html
https://www.mysqltutorial.org/mysql-stored-procedure/mysql-while-loop/
https://www.mysqltutorial.org/mysql-if-statement
https://www.javatpoint.com/mysql-trigger
https://www.javatpoint.com/mysql-trigger
https://www.w3resource.com/mysql/mysql-triggers.php