freelance_Project available to buy contact on 8007592194

SR.NO	Project NAME	Technology
1	Online E-Learning Platform Hub	React+Springboot+MySql
2	PG Mates / RoomSharing / Flat Mates	React+Springboot+MySql
3	Tour and Travel management System	React+Springboot+MySql
4	Election commition of India (online Voting System)	React+Springboot+MySql
5	HomeRental Booking System	React+Springboot+MySql
6	Event Management System	React+Springboot+MySql
7	Hotel Management System	React+Springboot+MySql
8	Agriculture web Project	React+Springboot+MySql
9	AirLine Reservation System / Flight booking System	React+Springboot+MySql
10	E-commerce web Project	React+Springboot+MySql
11	Hospital Management System	React+Springboot+MySql
12	E-RTO Driving licence portal	React+Springboot+MySql
13	Transpotation Services portal	React+Springboot+MySql
14	Courier Services Portal / Courier Management System	React+Springboot+MySql
15	Online Food Delivery Portal	React+Springboot+MySql
16	Muncipal Corporation Management	React+Springboot+MySql
17	Gym Management System	React+Springboot+MySql
18	Bike/Car ental System Portal	React+Springboot+MySql
19	CharityDonation web project	React+Springboot+MySql
20	Movie Booking System	React+Springboot+MySql

freelance_Project available to buy contact on 8007592194

21	Job Portal web project	React+Springboot+MySql
22	LIC Insurance Portal	React+Springboot+MySql
23	Employee Management System	React+Springboot+MySql
24	Payroll Management System	React+Springboot+MySql
25	RealEstate Property Project	React+Springboot+MySql
26	Marriage Hall Booking Project	React+Springboot+MySql
27	Online Student Management portal	React+Springboot+MySql
28	Resturant management System	React+Springboot+MySql
29	Solar Management Project	React+Springboot+MySql
30	OneStepService LinkLabourContractor	React+Springboot+MySql
31	Vehical Service Center Portal	React+Springboot+MySql
32	E-wallet Banking Project	React+Springboot+MySql
33	Blogg Application Project	React+Springboot+MySql
34	Car Parking booking Project	React+Springboot+MySql
35	OLA Cab Booking Portal	React+Springboot+MySql
36	Society management Portal	React+Springboot+MySql
37	E-College Portal	React+Springboot+MySql
38	FoodWaste Management Donate System	React+Springboot+MySql
39	Sports Ground Booking	React+Springboot+MySql
40	BloodBank mangement System	React+Springboot+MySql
41	Bus Tickit Booking Project	React+Springboot+MySql
42	Fruite Delivery Project	React+Springboot+MySql
43	Woodworks Bed Shop	React+Springboot+MySql
44	Online Dairy Product sell Project	React+Springboot+MySql
45	Online E-Pharma medicine sell Project	React+Springboot+MySql
46	FarmerMarketplace Web Project	React+Springboot+MySql
47	Online Cloth Store Project	React+Springboot+MySql
48		React+Springboot+MySql
49		React+Springboot+MySql
50		React+Springboot+MySql

https://www.youtube.com/@codewitharrays

https://www.instagram.com/codewitharrays/

https://t.me/codewitharrays Group Link: https://t.me/cceesept2023

+91 8007592194 +91 9284926333

codewitharrays@gmail.com

https://codewitharrays.in/project

DBMS – DATABASE MANAGEMENT SYSTEM

• Any enterprise application need to manage data.
• In early days of software development, programmers store data into files and does operation on it. However data is highly application specific.
• Even today many software manage their data in custom formats e.g. Tally, Address book, etc.
• As data management became more common, DBMS systems were developed to handle the data. This enabled developers to focus on the business logic e.g. FoxPro, DBase, Excel, etc.
• At least CRUD (Create, Retrieve, Update and Delete) operations are supported by all databases.
• Traditional databases are file based, less secure, single-user, non-distributed, manage less amount of data (MB), complicated relation management, file-locking and need number of lines of code to use in applications.

RDBMS – RELATIONAL DATABASE MANAGEMENT SYSTEM.

• RDBMS is relational DBMS.
• It organizes data into Tables, rows and columns. The tables are related to each other.
• RDBMS follow table structure, more secure, multi-user, server-client architecture, server side processing, clustering support, manage huge data (TB), built-in relational capabilities, table-locking or row-locking and can be easily integrated with applications.
• e.g. DB2, Oracle, MS-SQL, MySQL, MS-Access, SQLite,
• RDBMS design is based on Codd's rules developed at IBM (in 1970).

SQL- STRUCTURED QUERY LANGUAGE

• Clients send SQL queries to RDBMS server and operations are performed accordingly.
• Originally it was named as RQBE (Relational Query By Example).
• SQL is ANSI standardised in 1987 and then revised multiple times adding new features. Recent revision in 2016.
• SQL is case insensitive.
• There are five major categories:
• DDL: Data Definition Language e.g. CREATE, ALTER, DROP, RENAME.
• DML: Data Manipulation Language e.g. INSERT, UPDATE, DELETE.
• DQL: Data Query Language e.g. SELECT.
• DCL: Data Control Language e.g. CREATE USER, GRANT, REVOKE.
• TCL: Transaction Control Language e.g. SAVEPOINT, COMMIT, ROLLBACK.
• Table & column names allows alphabets, digits & few special symbols.
• If name contains special symbols then it should be back-quotes.

- e.g. Tbl1, `T1#`, `T2$` etc. Names can be max 30 chars long.

MySQL

- Developed by Michael Widenius in 1995. It is named after his daughter name Myia.
- Sun Microsystems acquired MySQL in 2008.
- Oracle acquired Sun Microsystem in 2010.
- MySQL is free and open-source database under GPL. However some enterprise modules are close sourced and available only under commercial version of MySQL.
- MariaDB is completely open-source clone of MySQL.
- MySQL support multiple database storage and processing engines.
- MySQL versions:
    - < 5.5: MyISAM storage engine
    - 5.5: InnoDb storage engine
    - 5.6: SQL Query optimizer improved, memcached style NoSQL
    - 5.7: Windowing functions, JSON data type added for flexible schema
    - 8.0: CTE, NoSQL document store.
- MySQL is database of year 2019 (in database engine ranking).

MySQL installation on Ubuntu/Linux

- terminal> sudo apt-get install mysql-community-server mysql-community-client
- This installs MySQL server (mysqld) and MySQL client (mysql).
- MySQL Server (mysqld)
    - Run as background process.
    - Implemented in C/C++.
    - Process SQL queries and generate results.
    - By default run on port 3306.
    - Controlled via systemctl.
    - terminal> sudo systemctl start|stop|status|enable|disable mysql
- MySQL client (mysql)
    - Command line interface
    - Send SQL queries to server and display its results.
    - terminal> mysql ±u root -p
- Additional MySQL clients
    - MySQL workbench
    - PHPMyAdmin

Getting started

- root login can be used to perform CRUD as well as admin operations.

- It is recommended to create users for performing non-admin tasks.
- mysql> CREATE DATABASE db;
- mysql> SHOW DATABASES;
- mysql> CREATE USER dbuser@localhostIDENTIFIED BY 'dbpass';
- mysql> SELECT user, host FROM mysql.user;
- mysql> GRANT ALL PRIVILEGES ON db.* TO dbuser@localhost;
- mysql> FLUSH PRIVILEGES;
- mysql> EXIT;
- terminal> mysql ±u dbuser ±pdbpass
- mysql> SHOW DATABASES;
- mysql> SELECT USER(), DATABASE();
- mysql> USE db;
- mysql> SHOW TABLES;
- mysql> CREATE TABLE student(id INT, name VARCHAR(20), marks DOUBLE);
- mysql> INSERT INTO student VALUES(1, 'Abc', 89.5);
- mysql> SELECT * FROM student;

Database logical layout

- Database/schema is like a namespace/container that stores all db objects related to a project.
- It contains tables, constraints, relations, stored procedures, functions, triggers, ...
- There are some system databases e.g. mysql, performance_schema, information_schema, sys, ... They contains db internal/system information.
    - e.g. SELECT user, host FROM mysql.user;
- A database contains one or more tables.
- Tables have multiple columns.
- Each column is associated with a data-type.
- Columns may have zero or more constraints.
- The data in table is in multiple rows.
- Each row have multiple values (as per columns).

Database physical layout

- In MySQL, the data is stored on disk in its data directory i.e. /var/lib/mysql
- Each database/schema is a separate sub-directory in data dir.
- Each table in the db, is a file on disk.
- e.g. student table in current db is stored in file /var/lib/mysql/db/student.ibd.
- Data is stored in binary format.

- A file may not be contiguously stored on hard disk.
- Data rows are not contiguous. They are scattered in the hard disk.
- In one row, all fields are consecutive.
- When records are selected, they are selected in any order

SQL scripts

- SQL script is multiple SQL queries written into a .sql file.
- SQL scripts are mainly used while database backup and restore operations.
- SQL scripts can be executed from terminal as:
    - terminal> mysql ±u user ±ppassword db < /path/to/sqlfile
- SQL scripts can be executed from command line as:
    - mysql> SOURCE /path/to/sqlfile
- Note that SOURCE is MySQL CLI client command.
- It reads commands one by one from the script and execute them on server

```
SHOW DATABASES;
Create database dacdb;
-- activate the database
     USE dacdb;

-- print current user & current database
     SELECT USER(), DATABASE();

-- print tables in current database
     SHOW TABLES;


CREATE TABLE students(id INT, name CHAR(20), marks DOUBLE);
SHOW TABLES;
INSERT INTO students VALUES (1, 'Nitin', 98.00);
INSERT INTO students VALUES (2, 'Sarang', 99.00);
INSERT INTO students VALUES (3, 'Nilesh', 77.00), (4, 'Sandeep',
88.00), (5, 'Amit', 90.00);
SELECT * FROM students;
```

Importing data into database

* Using SOURCE command.
    * classwork-db.sql --> dacdb database.

```SQL
-- SOURCE /path/to/the/sql/file
SOURCE D:\pgdiploma\edac-dbt\data\classwork-db.sql

SHOW TABLES;
SELECT * FROM books;
```

MySQL data types

- RDBMS have similar data types (but not same).
- MySQL data types can be categorised as follows
- **Numeric types (Integers)**
    - TINYINT (1 byte), SMALLINT (2 byte), MEDIUMINT (3 byte), INT (4 byte), BIGINT (8 byte), BIT(n bits)
    - integer types can signed (default) or unsigned.
- **Numeric types (Floating point)**
    - approx. precision ± FLOAT (4 byte), DOUBLE (8 byte) | DECIMAL(m, n) ± exact precision
- **Date/Time types**
    - DATE, TIME, DATETIME, TIMESTAMP, YEAR
- **String types ± size = number of chars * size of char**
    - CHAR(1-255) ± Fixed length, Very fast access.
    - VARCHAR(1-65535) ± Variable length, Stores length + chars.
    - TINYTEXT (255), TEXT (64K), MEDIUMTEXT (16M), LONGTEXT (4G) ± Variable length, Slower access.
- **Binary types ± size = number of bytes**
    - BINARY, VARBINARY, TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB
- **Miscellaneous types**
    - ENUM, SET
- **Size of char depends on Charset :-**
    - ASCII :- 1 Byte
    - Unicode :- 2 Byte
    - EBCDZF :- 4 Byte

CHAR vs VARCHAR vs TEXT

- **CHAR**
  - Fixed inline storage.
  - If smaller data is given, rest of space is unused.
  - Very fast access.
- **VARCHAR**
  - Variable inline storage.
  - Stores length and characters.
  - Slower access than CHAR.
- **TEXT**
  - Variable external storage.
  - Very slow access.
  - Not ideal for indexing.
- CREATE TABLE temp(c1 CHAR(4), c2 VARCHAR(4), c3 TEXT(4));
- DESC temp;
- INSERT INTO temp VALUES('abcd', 'abcd', 'abcdef');

DDL (Data Definition Language):

DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.DDL is a set of SQL commands used to create, modify, and delete database structures but not data. These commands are normally not used by a general user, who should be accessing the database via an application.

List of DDL commands:

- **CREATE**: This command is used to create the database or its objects (like table, index, function, views, store procedure, and triggers).
- **DROP**: This command is used to delete objects from the database.
- **ALTER**: This is used to alter the structure of the database.
- **TRUNCATE**: This is used to remove all records from a table, including all spaces allocated for the records are removed.
- **COMMENT**: This is used to add comments to the data dictionary.
- **RENAME**: This is used to rename an object existing in the database.

```
## DDL - CREATE TABLE
* CREATE TABLE tablename(col1 COL-TYPE, col2 COL-TYPE, col3 COL-TYPE,
...);
* CREATE TABLE tablename(col1 COL-TYPE constraint, col2 COL-TYPE
constraint, col3 COL-TYPE constraint, ..., constraint);
```

```SQL
CREATE TABLE test(c1 CHAR(10), c2 VARCHAR(10), c3 TEXT(10));
INSERT  INTO test VALUES ('ABCD', 'ABCD', 'ABCD');
SELECT * FROM test;

INSERT INTO test VALUES ('abcdefghijk', 'abcdefghijk', 'abcdefghijk');
-- error: data too long for c1.
INSERT INTO test VALUES ('abcdefghij', 'abcdefghijk', 'abcdefghijk');
-- error: data too long for c2.
INSERT INTO test VALUES ('abcdefghij', 'abcdefghij', 'abcdefghijk');
-- no error: upto 255 chars allowed in tinytext

SELECT * FROM test;
```

DQL (Data Query Language):

DQL statements are used for performing queries on the data within schema objects. The purpose of the DQL Command is to get some schema relation based on the query passed to it. We can define DQL as follows it is a component of SQL statement that allows getting data from the database and imposing order upon it. It includes the SELECT statement. This command allows getting the data out of the database to perform operations with it. When a SELECT is fired against a table or tables the result is compiled into a further temporary table, which is displayed or perhaps received by the program i.e. a front-end.

List of DQL:

- **SELECT**: It is used to retrieve data from the database.
    - `SELECT * FROM test;`

SELECT ± DQL

• Select all columns (in fixed order).
    • SELECT * FROM table;
• Select specific columns / in arbitrary order.
    • SELECT c1, c2, c3 FROM table;
• Column alias
    • SELECT c1 AS col1, c2 col2 FROM table;
• Computed columns.
    • SELECT c1, c2, c3, expr1, expr2 FROM table;
    • SELECT c1, • CASE WHEN condition1 THEN value1,
    • CASE WHEN condition2 THEN value2,
    • ELSE valuen
    • END
    • FROM table;

- Distinct values in column.
    - SELECT DISTINCT c1 FROM table;
    - SELECT DISTINCT c1, c2 FROM table;
- Select limited rows.
    - SELECT * FROM table LIMIT n;
    - SELECT * FROM table LIMIT m, n;

SELECT ± DQL ± ORDER BY

- In db rows are scattered on disk. Hence may not be fetched in a fixed order.
- Select rows in asc order.
    - SELECT * FROM table ORDER BY c1;
    - SELECT * FROM table ORDER BY c2 ASC;
- Select rows in desc order.
    - SELECT * FROM table ORDER BY c3 DESC;
- Select rows sorted on multiple columns.
    - SELECT * FROM table ORDER BY c1, c2;
    - SELECT * FROM table ORDER BY c1 ASC, c2 DESC;
    - SELECT * FROM table ORDER BY c1 DESC, c2 DESC;
- Select top or bottom n rows.
    - SELECT * FROM table ORDER BY c1 ASC LIMIT n;
    - SELECT * FROM table ORDER BY c1 DESC LIMIT n;
    - SELECT * FROM table ORDER BY c1 ASC LIMIT m, n

SELECT ± DQL ± WHERE

- It is always good idea to fetch only required rows (to reduce network traffic).
- The WHERE clause is used to specify the condition, which records to be fetched.
- Relational operators
    - , <=, >=, =, != or <>
- NULL related operators
    - NULL is special value and cannot be compared using relational operators.
    - IS NULL or <=>, IS NOT NULL.
- Logical operators
    - AND, OR, NOT
- BETWEEN operator (include both ends)
    - c1 BETWEEN val1 AND val2
- IN operator (equality check with multiple values)
    - c1 IN (val1, val2, val3)
- LIKE operator (similar strings)
- c1 LIKE “pattern”.
- % represent any number of any characters.
- _ represent any single character

DQL - SELECT

```SQL
SELECT * FROM books;
SELECT id, subject, price, author, name FROM books;
SELECT id, name, price FROM books;
SELECT * FROM emp;

-- fetch emp id, name and sal from emp table.
     SELECT empno, ename, sal FROM emp;

     SELECT  empno AS 'emp id', ename AS 'emp name', sal AS 'salary'
     FROM emp;

-- AS keyword is used to give alias to a column.
-- AS keyword is optional.
-- if alias name contains space or special chars, they must be
quoted. '---' or `---`
-- if alias name doesn't contain space or special chars, quotes are
optional.
     SELECT  empno 'emp id', ename 'emp name', sal salary FROM emp;
```

Computed column

```
-- print book id, name, price and gst (5% of price).
```

- `SELECT id, name, price FROM books;`
- `SELECT id, name, price, price * 0.05 FROM books;`
- `SELECT id, name, price, price * 0.05 gst FROM books;`

```
-- print book id, name, price, gst (5% of price) and total (price +
gst).
     SELECT id, name, price, price * 0.05 gst, price + price * 0.05
     total FROM books;

-- print empno, ename, sal, category of employee (emp table).
-- <= 1500: Poor, > 1500 AND <= 2500: Middle, > 2500: Rich
     SELECT empno, ename, sal FROM emp;
     SELECT empno, ename, sal,
     CASE
     WHEN sal <= 1500 THEN 'Poor'
     WHEN sal > 1500 AND sal <= 2500 THEN 'Middle'
```

```
    ELSE 'Rich'
    END AS category
    FROM emp;

-- print empno, ename, sal, category of employee (emp table) and dept.
10=ACCOUNTS, 20=RESEARCH, 30=SALES, *=OPERATIONS
    SELECT empno, ename, sal,
    CASE
    WHEN sal <= 1500 THEN 'Poor'
    WHEN sal > 1500 AND sal <= 2500 THEN 'Middle'
    ELSE 'Rich'
    END AS category,
    deptno,
    CASE
    WHEN deptno=10 THEN 'ACCOUNTS'
    WHEN deptno=20 THEN 'RESEARCH'
    WHEN deptno=30 THEN 'SALES'
    ELSE 'OPERATIONS'
    END AS dept
    FROM emp;
```

```
## DISTINCT column
SELECT subject FROM books;

-- fetch unique subjects from books
    SELECT DISTINCT subject FROM books;

-- fetch unique deptno from emp.
    SELECT DISTINCT deptno FROM emp;

-- fetch unique job from emp.
    SELECT DISTINCT job FROM emp;

-- unique jobs per dept OR unique depts per job OR unique combination
of dept & job.
    SELECT DISTINCT deptno, job FROM emp;
```

LIMIT clause

* SELECT cols FROM tablename LIMIT n;
    * get n rows.
* SELECT cols FROM tablename LIMIT m,n;
    * get n rows after skipping first m rows.

```SQL
SELECT * FROM books;
-- get first 5 books
     SELECT * FROM books LIMIT 5;
-- skip first 3 books and get next 2 books
     SELECT * FROM books LIMIT 3,2;
```

ORDER BY clause

* SELECT cols FROM tablename ORDER BY col;
    * sort records by column in asc order. (default order is ASC)
* SELECT cols FROM tablename ORDER BY col ASC;
    * sort records by column in asc order.
* SELECT cols FROM tablename ORDER BY col DESC;
    * sort records by column in desc order.

```SQL
  - SELECT * FROM books ORDER BY price;
  - SELECT * FROM books ORDER BY price DESC;
  - SELECT * FROM books ORDER BY author;
  - SELECT * FROM emp ORDER BY hire;
  - SELECT empno,ename,deptno,job FROM emp ORDER BY deptno,job;
  - SELECT empno,ename,deptno,job FROM emp ORDER BY job,deptno;
  - SELECT deptno,job FROM emp ORDER BY deptno,job;

-- sort all emp deptwise (asc), for same depts sort salwise in desc
order.
     SELECT * FROM emp ORDER BY deptno ASC, sal DESC;

-- in mysql, ORDER BY can be done by alias name.
     SELECT empno, ename, sal,
     CASE
     WHEN sal <= 1500 THEN 'Poor'
     WHEN sal > 1500 AND sal <= 2500 THEN 'Middle'
```

```
    ELSE 'Rich'
    END AS category
    FROM emp
    ORDER BY category;

-- in mysql, ORDER BY can be done by column number.
    SELECT empno, ename, sal,
    CASE
    WHEN sal <= 1500 THEN 'Poor'
    WHEN sal > 1500 AND sal <= 2500 THEN 'Middle'
    ELSE 'Rich'
    END AS category
    FROM emp
    ORDER BY 4 DESC;

## ORDER BY + LIMIT

-- print book with highest price
    SELECT * FROM books ORDER BY price DESC LIMIT 1;

-- print book with lowest price
    SELECT * FROM books ORDER BY price ASC LIMIT 1;

-- print book with third highest price
    SELECT * FROM books ORDER BY price DESC;
    SELECT * FROM books ORDER BY price DESC LIMIT 2,1;
```

WHERE clause

* SELECT cols FROM tablename WHERE condition;
    - only rows matching condition (=true) will be displayed.
* Relational operators
    - <, >, <=, =>, =, != or <>
* Logical operators
    - AND, OR, NOT

```SQL
  • SELECT * FROM emp WHERE deptno=20;
  • SELECT * FROM emp WHERE sal > 2500;
  • SELECT * FROM emp WHERE job = 'SALESMAN';
```

- `SELECT * FROM emp WHERE job = 'ANALYST' AND deptno = 20;`

```
```

- `SELECT DISTINCT deptno,job FROM emp ORDER BY deptno,job;`
- `SELECT * FROM emp ORDER BY 1,2,3,4,5,6,7,8;`

DML(Data Manipulation Language):

The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements. It is the component of the SQL statement that controls access to data and to the database. Basically, DCL statements are grouped with DML statements.

List of DML commands:

- **INSERT** : It is used to insert data into a table.
- **UPDATE:** It is used to update existing data within a table.
- **DELETE** : It is used to delete records from a database table.
- **LOCK:** Table control concurrency.
- **CALL:** Call a PL/SQL or JAVA subprogram.
- **EXPLAIN PLAN:** It describes the access path to data.

INSERT ± DML

- Insert a new row (all columns, fixed order).
  - INSERT INTO table VALUES (v1, v2, v3);
- Insert a new row (specific columns, arbitrary order).
  - INSERT INTO table(c3, c1, c2) VALUES (v3, v1, v2);
  - INSERT INTO table(c1, c2) VALUES (v1, v2);
  - Missing columns data is NULL.
  - NULL is special value and it is not stored in database.
- Insert multiple rows.
  - INSERT INTO table VALUES (av1, av2, av3), (bv1, bv2, bv3), (cv1, cv2, cv3).
- Insert rows from another table.
  - INSERT INTO table SELECT c1, c2, c3 FROM another-table;
  - INSERT INTO table (c1,c2) SELECT c1, c2 FROM another-table;

```
## DML - INSERT
* INSERT INTO tablename VALUES (v1, v2, v3, ...);
    * Strings and Date/Time must be enclosed in single quotes.
    * Other values without single quotes.
    * VALUES order must be same as of column order (while creating
table).
```

```SQL
INSERT INTO students VALUES (6, 'yogesh', 90);

INSERT INTO students VALUES (90, 7, 'digvijay');
-- error
INSERT INTO students(marks,id,name) VALUES (90, 7, 'digvijay');

SELECT * FROM students;
INSERT INTO students(id,name) VALUES (8, 'pooja');
INSERT INTO students(id,name) VALUES (9, 'sameer'), (10, 'shekhar'),
(11, 'rahul');
INSERT INTO students(id,name,marks) VALUES (12, NULL, NULL);
SELECT * FROM students;
```

* We can insert records from one table into another table.
    * INSERT INTO newtablename SELECT * FROM oldtablename;
        * The count and order of columns in newtable must be same as oldtable.
    * INSERT INTO newtablename(c1,c2) SELECT c1,c2 FROM oldtablename;
        * Column c1 and c2 data from oldtable will be inserted into c1 & c2 Columns of newtable.

```SQL
CREATE TABLE newstudents (roll INT, name CHAR(20));
INSERT INTO newstudents(roll,name) SELECT id,name FROM students;
SELECT * FROM newstudents;
```

Assignment Discussion

```
USE sales;
```

- select * from orders;

```
-- 10 rows
+------+---------+------------+------+------+
| onum | amt     | odate      | cnum | snum |
+------+---------+------------+------+------+
| 3001 |   18.69 | 1990-10-03 | 2008 | 1007 |
| 3003 |  767.19 | 1990-10-03 | 2001 | 1001 |
```

```
| 3002 | 1900.10 | 1990-10-03 | 2007 | 1004 |
| 3005 | 5160.45 | 1990-10-03 | 2003 | 1002 |
+------+---------+------------+------+------+
```

- `select * from orders WHERE odate = '1990-10-03';`

```
+------+---------+------------+------+------+
| onum | amt     | odate      | cnum | snum |
+------+---------+------------+------+------+
| 3001 |   18.69 | 1990-10-03 | 2008 | 1007 |
| 3003 |  767.19 | 1990-10-03 | 2001 | 1001 |
+------+---------+------------+------+------+
```

- `select * from orders WHERE odate = '1990-10-03' AND cnum > 2003;`

```
+------+---------+------------+------+------+
| onum | amt     | odate      | cnum | snum |
+------+---------+------------+------+------+
| 3001 |   18.69 | 1990-10-03 | 2008 | 1007 |
| 3002 | 1900.10 | 1990-10-03 | 2007 | 1004 |
| 3006 | 1098.16 | 1990-10-03 | 2008 | 1007 |
+------+---------+------------+------+------+
```

- `select * from orders WHERE NOT (odate = '1990-10-03' AND cnum > 2003);`

```
+------+---------+------------+------+------+
| onum | amt     | odate      | cnum | snum |
+------+---------+------------+------+------+
| 3003 |  767.19 | 1990-10-03 | 2001 | 1001 |
| 3005 | 5160.45 | 1990-10-03 | 2003 | 1002 |
| 3009 | 1713.23 | 1990-10-04 | 2002 | 1003 |
+------+---------+------------+------+------+
```

- `select * from orders WHERE amt < 1000;`

```
+------+--------+------------+------+------+
| onum | amt    | odate      | cnum | snum |
+------+--------+------------+------+------+
| 3001 |  18.69 | 1990-10-03 | 2008 | 1007 |
| 3003 | 767.19 | 1990-10-03 | 2001 | 1001 |
| 3007 |  75.75 | 1990-10-04 | 2004 | 1002 |
| 3010 | 309.95 | 1990-10-04 | 2004 | 1002 |
+------+--------+------------+------+------+
```

- select * from orders WHERE amt < 1000 OR NOT (odate = '1990-10-03' AND cnum > 2003);

```
+------+---------+------------+------+------+
| onum | amt     | odate      | cnum | snum |
+------+---------+------------+------+------+
| 3001 |  18.69 | 1990-10-03 | 2008 | 1007 |
| 3003 |  767.19 | 1990-10-03 | 2001 | 1001 |
| 3005 | 5160.45 | 1990-10-03 | 2003 | 1002 |
| 3009 | 1713.23 | 1990-10-04 | 2002 | 1003 |
+------+---------+------------+------+------+
```

```
-- sales
-- Write a query on the Customers table whose output will exclude all customers with a rating <= 100, unless they are located in Rome.
USE sales;
```

- SELECT * FROM customers;
- SELECT * FROM customers WHERE rating <= 100;
- SELECT * FROM customers WHERE rating <= 100 AND city != 'Rome';
- SELECT * FROM customers WHERE NOT(rating <= 100 AND city != 'Rome');

WHERE vs WHEN

* WHERE is to fetch selected rows -- for which given condition is true
-- after FROM tablename.
* CASE...WHEN is for computed column -- before FROM tablename.

```SQL
    USE dacdb;

    SELECT empno, ename, sal,
    CASE
    WHEN sal <= 1500 THEN 'Poor'
    WHEN sal > 1500 AND sal <= 2500 THEN 'Middle'
    ELSE 'Rich'
    END AS category
    FROM emp
    WHERE job = 'SALESMAN';
```

NULL related operators

* NULL doesn't work with relational and logical operators.
* Need special operators
    * IS NULL or <=>
    * IN NOT NULL

```SQL
SELECT * FROM emp;

-- find all emps whose comm is null.
        • SELECT * FROM emp WHERE comm = NULL;
        • SELECT * FROM emp WHERE comm IS NULL;
        • SELECT * FROM emp WHERE comm <=> NULL;

-- find all emps whose comm is not null.
        • SELECT * FROM emp WHERE comm IS NOT NULL;
```

BETWEEN operator

* more readable for comparing within range (than AND).
* faster than AND operator for same task.
* NOT BETWEEN

```SQL
-- get all emps whose sal is between 1500 and 2000.
        • SELECT * FROM emp WHERE sal >= 1500 AND sal <= 2000;
        • SELECT * FROM emp WHERE sal BETWEEN 1500 AND 2000;

-- get all emps hired in year 1982.
        • SELECT * FROM emp WHERE hire BETWEEN '1982-01-01' AND '1982-
          12-31';

-- get all emps whose name is between 'F' to 'K'.
        • INSERT INTO emp(empno, ename) VALUES (1, 'F'), (2, 'K'), (3,
          'L');
        • SELECT * FROM emp WHERE ename BETWEEN 'F' AND 'K';
```

```
-- will take all emp whose name starts between F and K. Also it includes name 'L' (if any).
```
- `SELECT * FROM emp WHERE ename BETWEEN 'F' AND 'L';`

```
-- will take all emp whose name starts between F and K. Skip name 'L' (if any).
```
- `SELECT * FROM emp WHERE ename BETWEEN 'F' AND 'L' AND ename != 'L';`

IN operator

* more readable for comparing equality with multiple values.
* faster than using OR for the same work.
* NOT IN operator

```SQL
-- find all ANALYST, MANAGER and PRESIDENT.
```
- `SELECT * FROM emp WHERE job='ANALYST' OR job='MANAGER' OR job='PRESIDENT';`

- `SELECT * FROM emp WHERE job IN ('ANALYST', 'MANAGER', 'PRESIDENT');`

LIKE operator

* find similar name
* special characters (wildcard characters)
    * '%' : any number of any characters
    * '_' : single any character
* NOT LIKE

```SQL
-- get all emps whose name start with B
```
- `SELECT * FROM emp WHERE ename LIKE 'B%';`

```
-- get all emps whose name end with H
```
- `SELECT * FROM emp WHERE ename LIKE '%H';`

```
-- get all emps whose name contains U.
```
- `SELECT * FROM emp WHERE ename LIKE '%U%';`

```
-- get all emps whose name contains any 4 letters.
  • SELECT * FROM emp WHERE ename LIKE '____';

-- find all emps whose name contains A twice.
  • SELECT * FROM emp WHERE ename LIKE '%A%A%';

-- find all emps whose name contains A only once.
  • SELECT * FROM emp WHERE ename LIKE '%A%' AND ename NOT LIKE
    '%A%A%';
```

```
# DQL - SELECT
    ## LIMIT clause
    ## ORDER BY clause
    ## WHERE clause
    ### BETWEEN, IN and LIKE
    * "%" means 0 or more occurrences of any character.

-- find all emps whose name contains I
  • SELECT empno, ename, sal FROM emp WHERE ename LIKE '%I%';

-- find all emps whose name contains L twice (consecutive).
  • SELECT empno, ename, sal FROM emp WHERE ename LIKE '%LL%';

-- find all emps whose name contains A twice.
  • SELECT empno, ename, sal FROM emp WHERE ename LIKE '%A%A%';

-- find all emps whose name contains A.
  • SELECT empno, ename, sal FROM emp WHERE ename LIKE '%A%';
Note:- emps name containa A once or multiple times.
  • INSERT INTO emp (empno,ename) VALUES (4, 'ANAMIKA');

-- find all emps whose name contains A exactly once.
  • SELECT empno, ename, sal FROM emp WHERE (ename LIKE '%A%') AND
    (ename NOT LIKE '%A%A%');

-- find all emps between 'F' and 'K'.
  • SELECT empno, ename, sal FROM emp WHERE ename BETWEEN 'F' AND
    'K';
```

```
-- final all emp names starting from F to K.
```
- `SELECT empno, ename, sal FROM emp WHERE ename BETWEEN 'F' AND 'K' OR ename LIKE 'K%';`
- `INSERT INTO emp (empno,ename) VALUES (5, 'ZEBRA');`

```
-- final all emp names starting from S to Z.
```
- `SELECT empno, ename, sal FROM emp WHERE ename BETWEEN 'S' AND 'Z' OR ename LIKE 'Z%';`

UPDATE ± DML

- To change one or more rows in a table.
- Update row(s) single column.
    - UPDATE table SET c2=new-value WHERE c1=some-value;
- Update multiple columns.
    - UPDATE table SET c2=new-value, c3=new-value WHERE c1=some-value;
- Update all rows single column.
    - UPDATE table SET c2=new-value;

```
# DML - UPDATE
    * UPDATE tablename SET colname=new-value WHERE colname=value;
```SQL
```
- `SELECT USER(), DATABASE();`
- `SHOW TABLES;`
- `SELECT * FROM books;`
- `UPDATE books SET price=223.450 WHERE id=1001;`
- `SELECT * FROM books;`

- `UPDATE books SET author='Yashwant P Kanetkar', price=323.450 WHERE id=1001;`
- `SELECT * FROM books;`

```
-- increase price of all 'C Programming' books by 10%.
```
- `UPDATE books SET price = price + price * 0.10 WHERE subject = 'C Programming';`

```
-- increase price of all books by 5%.
```
- `UPDATE books SET price = price + price * 0.05;`

```

DELETE ± DML vs TRUNCATE ± DDL vs DROP ± DDL

- **DELETE**
  - To delete one or more rows in a table.
  - Delete row(s)
    - DELETE FROM table WHERE c1=value;
  - Delete all rows
    - DELETE FROM table :-
      1] Mark all rows as deleted.
      2]Space occupied by then can be overwritten/resued by new records.
      3]Actual table file size is not changed(much)
      4]DML query-roll backed.
- **TRUNCATE**
  - Delete all rows.
    - TRUNCATE TABLE table; :-
      1]Truncate file size so that only structure is kept.
      2]All rows space is released.
      3]Much faster operation for huge table.
      4]DDL query-can never be roll backed.
  - Truncate is faster than DELETE.
- **DROP**
  - Delete all rows as well as table structure.
    - DROP TABLE table; :--
      1] Delete table file structure and data
      2]DDL- no rollback
    - DROP TABLE table IF EXISTS;
  - Delete database/schema.
    - DROP DATABASE db


```
# DML - DELETE
* Can delete one or more rows.
* DELETE FROM tablename WHERE condition;
```SQL
-- delete single row
```

- `DELETE FROM books WHERE id=1001;`
- `SELECT * FROM books;`
```

```
-- delete multiple rows
   • DELETE FROM books WHERE subject='C++ Programming';
   • SELECT * FROM books;

-- delete all rows
   • DELETE FROM books;
   • DESCRIBE books;
   • SELECT * FROM books;
```

DDL - TRUNCATE

```
-- delete all rows
   • TRUNCATE TABLE emp;
   • DESCRIBE emp;
   • SELECT * FROM emp;
```

DDL - DROP TABLE

* Deletes table structure as well as table data.

```SQL
   • SELECT * FROM dept;
   • DROP TABLE dept;
   • DESCRIBE dept;
   • SELECT * FROM dept;
```

```SQL
   • SHOW TABLES;
   • DROP TABLE items;
   • DROP TABLE IF EXISTS items;
   • DROP TABLE IF EXISTS salgrade;
   • SHOW TABLES;

```

Seeking HELP

- HELP is client command to seek help on commands/functions.
  - HELP SELECT;
  - HELP Functions;
  - HELP SIGN;

Restore all tables back

```SQL
  - SOURCE D:/pgdiploma/edac-dbt/data/classwork-db.sql
```

HELP

```SQL
  - HELP SELECT;
  - HELP Functions;
  - SELECT SUBSTRING(ename,1,1), ename FROM emp;
  - SELECT empno, ename, sal FROM emp WHERE SUBSTRING(ename,1,1)
    BETWEEN 'F' AND 'K';
```

DUAL table

- First used in oracle. Two rows table later it made single row & col table.
- A dummy/in-memory a table having single row & single column.
- It is used for arbitrary calculations, testing functions, etc.
  - SELECT 2 + 3 * 4 FROM DUAL;
  - SELECT NOW() FROM DUAL;
  - SELECT USER(), DATABASE() FROM DUAL;
- In MySQL, DUAL keyword is optional.
  - SELECT 2 + 3 * 4;
  - SELECT NOW();
  - SELECT USER(), DATABASE();

DUAL Table

* ANSI optional keywords: AS, ASC, DUAL.

```SQL
SHOW TABLES;
SELECT 2 + 3 * 4 FROM DUAL;
DESCRIBE DUAL;
-- error
```

```SQL
SELECT * FROM DUAL;
-- error
SELECT 2 + 3 * 4;
```

Numeric & String functions

• ABS() • POWER() • ROUND(), FLOOR(), CEIL()
• ASCII(), CHAR()
• CONCAT()
• SUBSTRING()
• LOWER(), UPPER()
• TRIM(), LTRIM(), RTRIM()
• LPAD(), RPAD() • REGEXP_LIKE()

SQL Functions

* Used to perform some operations on table data.
* Can also be used without any table data with aribitrary values with optional DUAL table.

```SQL
HELP Functions;
```

Information Functions

```SQL
-- get current user name
   • SELECT USER() FROM DUAL;
-- get current database/schema
   • SELECT DATABASE() FROM DUAL;
-- get current server date & time
   • SELECT NOW() FROM DUAL;
-- get current server date & time
   • SELECT SYSDATE() FROM DUAL;
-- get server version
   • SELECT VERSION() FROM DUAL;
```

String Functions

```SQL
- HELP String Functions;
- HELP ASCII;
- SELECT ASCII('A'), ASCII('a'), ASCII('0'), ASCII(' ');

- HELP CHAR Function;
- SELECT CHAR(65 USING ASCII);
- SELECT LENGTH('Sunbeam'), LENGTH('Infotech');
- SELECT ename, LENGTH(ename) FROM emp;

-- find all names which contains 4 characters
- SELECT ename FROM emp WHERE ename LIKE '____';
- SELECT ename FROM emp WHERE LENGTH(ename) = 4;

- SELECT CONCAT('Hello', ' ', 'World');
- SELECT CONCAT('Hello', ' ', 12345);

-- show message: XYZ employee is working as ABC on salary PQR in
department 10.
- SELECT CONCAT(ename, ' employee is working as ', job, ' on salary
  ', sal, ' in department ', deptno) AS msg FROM emp;

-- print all book names
- SELECT name FROM books;

HELP SUBSTRING;
-- SUBSTRING(string, position, length)
-- position (start from 1)
--  +ve: from start of string
--  -ve: from end of string
-- length (optional)
--  if length is not given, till end of string.
--  +ve: number of characters from the given position.
--  0 or -ve: no meaning -- results in empty string.
-- MySQL specific:
--  allows FROM keyword to indicate position and FOR keyword to
indicate length.
--  not allowed with all other RDBMS.
--  not commonly used syntax.
```

```
-- print first 4 letters of name.
   • SELECT name, SUBSTRING(name, 1, 4) FROM books;

-- print book name 4th letter onwards.
   • SELECT name, SUBSTRING(name, 4) FROM books;

-- print last 4 letters of name
   • SELECT name, SUBSTRING(name, -4) FROM books;
   • SELECT SUBSTRING('Sunbeam', 4, -2);

-- find all names starting from 'F' to 'K'.
   • SELECT ename FROM emp WHERE SUBSTRING(ename, 1, 1) BETWEEN 'F'
     AND 'K';

*LEFT(), RIGHT(), MID(), SUBSTR()

-- print first 2 chars and last 2 chars of book name
   • SELECT LEFT(name,2), RIGHT(name,2) FROM books;

-- LTRIM(), RTRIM(), TRIM() -- remove leading (left) or trailing
(right) white-spaces (space, tab, newline)
   • SELECT TRIM('    ABCD          ');

-- LPAD(), RPAD() -- add given char at start/end.
   • SELECT LPAD('SunBeam', 10, '*');
   • SELECT RPAD('SunBeam', 10, '*');

   • SELECT LPAD('SunBeam', 12, '*');
   • SELECT RPAD(LPAD('SunBeam', 12, '*'), 17, '*');
-- result of LPAD() is passed as first argument to RPAD().

-- REPLACE() -- find occurence of a word and replace with other
   • SELECT REPLACE('this', 'is', 'at');
   • SELECT REPLACE('this is a string', 'is', 'at');
   • SELECT REPLACE('this is a string', ' ', '');

-- UPPER(), LOWER()
   • SELECT name, UPPER(name), LOWER(name) FROM books;
```

Date-Time and Information functions

- VERSION() • USER(), DATABASE()
- MySQL supports multiple date time related data types
- DATE (3), TIME (3), DATETIME (5), TIMESTAMP (4), YEAR (1)
- SYSDATE(), NOW() • DATE(), TIME()
- DAYOFMONTH(), MONTH(), YEAR(), HOUR(), MINUTE(), SECOND(),
- DATEDIFF(), DATE_ADD(), TIMEDIFF()
- MAKEDATE(), MAKETIME()

Date and Time Functions

```SQL
HELP Date and Time Functions;

-- NOW() & SYSDATE() returns current date & time.
SLEEP() holds query execution for given seconds. No output/result.
   • SELECT NOW(), SLEEP(5), SYSDATE();

-- return server date, time and date-time respectively.
CURRENT_TIMESTAMP() is similar to NOW() / SYSDATE().
   • SELECT CURRENT_DATE(), CURRENT_TIME(), CURRENT_TIMESTAMP();
   • SELECT CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP;

-- DAYOFYEAR(), DAYOFMONTH(), MONTH(), MONTHNAME(), YEAR(), HOUR(),
MINUTE(), SECOND()
   • SELECT DAYOFMONTH(NOW()), MONTH(NOW()), MONTHNAME(NOW()),
     YEAR(NOW()), DAYOFYEAR(NOW());

-- DATE() and TIME() component of DATETIME
   • SELECT DATE(NOW()), TIME(NOW());

HELP DATE_ADD;
HELP DATEDIFF;
HELP TIMESTAMPDIFF;

--
SELECT DATE_ADD('2020-10-27', INTERVAL 1 DAY);
SELECT DATE_ADD(NOW(), INTERVAL 1 DAY);
SELECT DATE_ADD(NOW(), INTERVAL 1 MONTH);
SELECT DATE_ADD(NOW(), INTERVAL 3 MONTH);
SELECT DATE_ADD(NOW(), INTERVAL 1 YEAR);
```

```
- first arg should be higher and second should be lower = +ve result
   • SELECT DATEDIFF(NOW(), '1983-09-28');

-- first arg lower and second higher = -ve result
   • SELECT DATEDIFF('1983-09-28', NOW());

-- first arg: difference unit, second arg: lower date, third arg:
higher date --> +ve result
   • SELECT TIMESTAMPDIFF(YEAR, '1983-09-28', NOW());

-- find ename, hire date and experience in months.
   • SELECT ename, hire, TIMESTAMPDIFF(MONTH, hire, NOW()) FROM emp;

   • SELECT ename, hire, TIMESTAMPDIFF(YEAR, hire, NOW()),
     TIMESTAMPDIFF(MONTH, hire, NOW()) FROM emp;
```

String Functions

```SQL
-- from start of string till first occurence of ' ' i.e. Sunbeam
   • SELECT SUBSTRING_INDEX('Sunbeam Infotech At Pune', ' ', 1);

-- from start of string till second occurence of ' ' i.e. Sunbeam
Infotech
   • SELECT SUBSTRING_INDEX('Sunbeam Infotech At Pune', ' ', 2);

-- from start of string till third occurence of ' ' i.e. Sunbeam
Infotech At
   • SELECT SUBSTRING_INDEX('Sunbeam Infotech At Pune', ' ', 3);

-- whole string because, ' ' have only three occurences.
   • SELECT SUBSTRING_INDEX('Sunbeam Infotech At Pune', ' ', 4);

-- from last occurence of ' ' till end i.e. Pune
   • SELECT SUBSTRING_INDEX('Sunbeam Infotech At Pune', ' ', -1);

-- from second last occurence of ' ' till end i.e. At Pune
   • SELECT SUBSTRING_INDEX('Sunbeam Infotech At Pune', ' ', -2);
```

Control Flow Functions

* IF(condition, expression_if_true, expression_if_false)

```SQL
   • HELP Control Flow Functions;
   • HELP IF Function;
   • SELECT * FROM books;

-- if book price is less than or equal to 500, not expensive, else
expensive.
   • SELECT name, price, IF(price <= 500, 'not expensive',
     'expensive') AS category FROM books;

-- if book price below 300, not expensive.
-- if book price above 300 to 600, moderate expensive.
-- if book price above 600, very expensive
   • SELECT name, price, IF(price < 300, 'not expensive', IF(price <=
     600, 'moderate expensive',  'very expensive') ) AS category FROM
     books;
   • INSERT INTO books VALUES (1, 'Atlas Shrugged', 'Any Rand',
     'Novell', 452.45);
   • SELECT subject FROM books;


-- if subject have atleast one space, result = from start of string to
that space.
-- if subject doesn't have single space, result is whole string.

   • SELECT subject, INSTR(subject, ' ') FROM books;
   • SELECT subject, LEFT(subject, INSTR(subject, ' ')) FROM books;
   • SELECT subject, IF(INSTR(subject, ' ')=0, subject, LEFT(subject,
     INSTR(subject, ' '))) first_word FROM books;
   • SELECT subject, SUBSTRING_INDEX(subject, ' ', 1) FROM books;
```

```
if(INSTR(subject, ' ') == 0) // space not found
    subject // take whole word
else
    LEFT(subject, INSTR(subject, ' ')) // take n chars from word
`-----------------------------------------------------------------
```

Numeric Functions
* FLOOR() -- returns immediately smaller whole number (int)
* CEIL() -- returns immediately higher whole number (int)

```SQL
  • HELP Numeric Functions;
  • SELECT POW(2, 4), POW(2, 10), POW(2, 0), POW(2, 0.5), POW(2, -1);
  • SELECT ABS(123), ABS(-123);
  • SELECT FLOOR(2.7), FLOOR(-2.7);
  • SELECT CEIL(2.4), CEIL(-2.4);

-- 123.2, 123.2346
  • SELECT ROUND(123.234567, 1), ROUND(123.234567, 4);

-- 12346, 12350, 12300
  • SELECT ROUND(12345.678, 0), ROUND(12345.678, -1),
    ROUND(12345.678, -2);

-- nearest tens -- 45 --> 50
  • SELECT ROUND(45.4, -1), ROUND(-45.4, -1);

-- nearest tens -- 44 --> 40
  • SELECT ROUND(44.4, -1), ROUND(-44.4, -1);
```

Control and NULL and List functions

• NULL is special value in RDBMS that represents absence of value in that column.
• NULL values do not work with relational operators and need to use special operators.
• Most of functions return NULL if NULL value is passed as one of its argument.
• ISNULL() • IFNULL() • NULLIF() • COALESCE() • GREATEST(), LEAST()
• IF(condition, true-value, false-value)

Null Operators and Functions
* NULL related Operators
    * IS NULL, <=>, IS NOT NULL
* NULL related Functions ()
    * ISNULL(), IFNULL(), NULLIF()
* ISNULL() returns 1 if NULL, else return 0.
* IFNULL(col, value) -- if col is null value, consider given 'value'
* NULLIF(col, value) -- if col value = given value, consider NULL.

* COALESCE(v1,v2,v3,...) -- returns first non-null value.

```SQL
  • SELECT comm, ISNULL(comm) FROM emp;
  • SELECT comm, IFNULL(comm, 0.0) FROM emp;


-- print emp name, sal, comm and total income.
  • SELECT ename, sal, comm, sal + comm AS income FROM emp;
  • SELECT ename, sal, comm, sal + IFNULL(comm,0.0) AS income FROM
    emp;
  • SELECT ename, sal, NULLIF(sal, 1500) FROM emp;

-- if comm is non-null, return it; otherwise return sal.
  • SELECT COALESCE(null, null, 12, 'abc');
  • SELECT comm, sal, COALESCE(comm,sal) FROM emp;
```

List Functions

* Functions take any number of arguments.
    * COALESCE()
    * CONCAT()
    * GREATEST()
    * LEAST()

```SQL
SELECT GREATEST(34, 56, 78, 12, 45); // output: 78

SELECT LEAST(34, 56, 78, 12, 45);   // output:- 12
```

Group functions

• Work on group of rows of table.

• Input to function is data from multiple rows & then output is single row. Hence these functions are called as "Multi Row FXQFWLRQ³ RU "Group FXQFWLRQV³.

• These functions are used to perform aggregate ops like sum, avg, max, min, count or std dev, etc. Hence these fns are also called as "Aggregate Functions".

• Example: SUM(), AVG(), MAX(), MIN(), COUNT().

• NULL values are ignored by group functions.

- Limitations of GROUP functions:
    - Cannot select group function along with a column.
    - Cannot select group function along with a single row fn.
    - Cannot use group function in WHERE clause/condition.
    - Cannot nest a group function in another group fn.

Group Functions

* Group Functions ignore NULL values.

```SQL
  • SELECT COUNT(empno), COUNT(comm) FROM emp;
  • SELECT SUM(sal), SUM(comm) FROM emp;
  • SELECT AVG(sal), AVG(comm) FROM emp;
  • SELECT MAX(sal), MAX(comm), MIN(sal), MIN(comm) FROM emp;
  • SELECT COUNT(comm), COUNT(IFNULL(comm,0)) FROM emp;
```

```SQL
-- error: limitation 1
SELECT ename, SUM(sal) FROM emp;


-- error: limitation 2
SELECT LOWER(ename), SUM(sal) FROM emp;

-- error: limitation 3
SELECT * FROM emp WHERE sal = MAX(sal);

-- error: limitation 4
SELECT COUNT(SUM(sal)) FROM emp;
```

Group Functions

```SQL
  • SELECT IFNULL(comm,0) FROM emp;

  • SELECT LEAST(sal, IFNULL(comm,0)) FROM emp;

  • SELECT MIN(LEAST(sal, IFNULL(comm,0))) FROM emp;
```

--error

```SQL
  • SELECT MIN(comm), LEAST(sal, IFNULL(comm,0)) FROM emp;
```

Limitations (SQL Limitations as per ANSI standard)
* Cannot select a column along with group function.
* Cannot select single row function along with group function.
* Cannot nest one group function in another group function.
* Cannot use group function in WHERE clause.

MySQL config
* Steps to ensure that GROUP BY and GROUP functions work as per RDBMS/SQL standards.
1. Go to C:\ProgramData\MySQL\MySQL Server 8.0.
2. Open file: my.ini using notepad or vscode.
3. line 108: sql-mode="ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION"
4. Restart the computer OR restart mysql server.
    * Run (Window+R) --> services.msc --> MySQL80 --> Right Click --> Stop and Right Click --> Start.

```SQL
    SELECT USER(), DATABASE();
    SELECT @@sql_mode;
```

GROUP BY clause

- GROUP BY is used for analysis of data i.e. generating reports & charts.
- When GROUP BY single column, generated output can be used to plot 2-D chart. When GROUP BY two column, generated output can be used to plot 3-D chart and so on.
- GROUP BY queries are also called as Multi-dimensional / Spatial queries.
- Syntactical Characteristics:
    - If a column is used for GROUP BY, then it may or may not be used in SELECT clause.
    - If a column is in SELECT, it must be in GROUP BY.
- When GROUP BY query is fired on database server, it does following:
    - Load data from server disk into server RAM.
    - Sort data on group by columns.
    - Group similar records by group columns.
    - Perform given aggregate ops on each column.
    - Send result to client

GROUP BY clause
* SELECT colname, GROUPFN(col2name) FROM tablename GROUP BY colname;

```SQL
-- error
   • SELECT deptno, COUNT(empno) FROM emp;

-- 14 emps = 3 emps (dept=10), 5 emps (dept=20), 6 emps (dept=30).
   • SELECT deptno, COUNT(empno) FROM emp GROUP BY deptno;
   • SELECT job, AVG(sal) FROM emp GROUP BY job;
```

GROUP BY
* terminal> mysql -u edac -pedac dacdb

```SQL
SHOW TABLES;

   • SELECT deptno, COUNT(empno) FROM emp GROUP BY deptno;
```

```
+--------+--------------+
| deptno | COUNT(empno) |
+--------+--------------+
|     10 |            3 |
|     20 |            5 |
|     30 |            6 |
+--------+--------------+
```

```
-- grouped column may not be in SELECT statement, however usually
result is meaningless/difficult understand.
   • SELECT COUNT(empno) FROM emp GROUP BY deptno;
```

```
+--------------+
| COUNT(empno) |
+--------------+
|            3 |
|            5 |
|            6 |
+--------------+
```

```
-- error: deptno is selected, then it must be grouped by.
   • SELECT deptno, COUNT(empno) FROM emp;
```

GROUP BY on multiple columns

```SQL
  • SELECT DISTINCT deptno, job FROM emp;
        -- 10, CLERK
        -- 10, MANAGER
        -- 10, PRESIDENT
        -- 20, CLERK
        -- 20, MANAGER
        -- 20, ANALYST
        -- 30, CLERK
        -- 30, MANAGER
        -- 30, SALESMAN

  • SELECT deptno, job, COUNT(empno) FROM emp
    GROUP BY deptno, job;
        -- 10, CLERK     --> 1
        -- 10, MANAGER   --> 1
        -- 10, PRESIDENT--> 1
        -- 20, CLERK     --> 2
        -- 20, MANAGER   --> 1
        -- 20, ANALYST   --> 2
        -- 30, CLERK     --> 1
        -- 30, MANAGER   --> 1
        -- 30, SALESMAN --> 4
```

GROUP BY with WHERE clause

* WHERE clause with filter records from the table.
* GROUP BY will group the records and perform aggregate operations.

```SQL
-- all 14 emps are sorted, grouped and sum(sal) is done.
  • SELECT job, SUM(sal) FROM emp
    GROUP BY job;

-- 11 emps (dept 20 & 30) emps are sorted, grouped and sum(sal) is
done.
  • SELECT job, SUM(sal) FROM emp
    WHERE deptno != 10
    GROUP BY job;
```

```
-- 8 emps (whose sal >= 1500) are sorted, grouped and sum(sal) is done.
   • SELECT job, SUM(sal) FROM emp
     WHERE sal >= 1500
     GROUP BY job;

   • SELECT job, SUM(sal) FROM emp
     WHERE job IN ('SALESMAN', 'ANALYST', 'MANAGER')
     GROUP BY job;
```

HAVING clause

* Used to filter result based on "aggregate function" calculation.
* HAVING must be used with GROUP BY and syntactically immediately after GROUP BY.
* HAVING is used to put condition based on aggregate function and/or grouped column only.
* However using HAVING clause on grouped columns slow down the execution. It is recommended to use WHERE clause.

```SQL
   • SELECT job, AVG(sal) FROM emp
     GROUP BY job;

-- find jobs whose avg sal is more than 1200.
   • SELECT job, AVG(sal) FROM emp
     WHERE AVG(sal) > 1200
     GROUP BY job;
Note: error: group functions cannot be used in WHERE clause

   • SELECT job, AVG(sal) FROM emp
     GROUP BY job
     HAVING AVG(sal) > 1200;

   • SELECT job, AVG(sal) FROM emp
     GROUP BY job
     HAVING sal > 1200;
Note: error: HAVING is used to put condition only on aggregate fns or grouped columns (not other columns.)
```

- ```
  SELECT job, SUM(sal) FROM emp
  WHERE job IN ('SALESMAN', 'ANALYST', 'MANAGER')
  GROUP BY job;
  ```

```
--+----------+----------+
--| job      | SUM(sal) |
--+----------+----------+
--| SALESMAN |  5600.00 |
--| MANAGER  |  8275.00 |
--| ANALYST  |  6000.00 |
--+----------+----------+
```

- ```
  SELECT job, SUM(sal) FROM emp
  GROUP BY job
  HAVING job IN ('SALESMAN', 'ANALYST', 'MANAGER');
  ```

```
--+----------+----------+
--| job      | SUM(sal) |
--+----------+----------+
--| SALESMAN |  5600.00 |
--| MANAGER  |  8275.00 |
--| ANALYST  |  6000.00 |
--+----------+----------+
```

GROUP BY with ORDER BY

```SQL
-- print avg sal per job in sorted order of job.
```

- ```
  SELECT job, AVG(sal) FROM emp
  GROUP BY job
  ORDER BY job;
  ```

- ```
  SELECT job, ROUND(AVG(sal), 2) FROM emp
  GROUP BY job
  ORDER BY job;
  ```

```
-- print avg sal per job in descending order of avg sal.
```

- ```
  SELECT job, AVG(sal) FROM emp
  GROUP BY job
  ORDER BY AVG(sal) DESC;
  ```
  ```

```
-- sort by 2nd column in SELECT.
   • SELECT job, AVG(sal) FROM emp
     GROUP BY job
     ORDER BY 2 DESC;

   • SELECT job, AVG(sal) AS avgsal FROM emp
     GROUP BY job
     ORDER BY avgsal DESC;
```

GROUP BY with ORDER BY and LIMIT

```SQL
-- find the dept which spend max/highest on sal of emps
   • SELECT deptno, SUM(sal) AS sumsal FROM emp
     GROUP BY deptno
     ORDER BY sumsal DESC
     LIMIT 1;

-- find the dept which spend second lowest on income (sal+comm) of
emps

   • SELECT deptno, SUM( sal + IFNULL(comm,0.0) ) AS sum_income FROM
     emp
     GROUP BY deptno
     ORDER BY sum_income ASC
     LIMIT 1,1;
```

SELECT syntax

```
HELP SELECT;

SELECT col1, expr1, expr2, ... FROM tablename
WHERE condition
GROUP BY col1, ...
HAVING condition
ORDER BY col1, ...
LIMIT m, n;
```---

Transaction

- Transaction is set of DML queries executed as a single unit.
- Transaction examples
    - accounts table [id, type, balance]
    - UPDATE accounts SET balance=balance-1000 WHERE id = 1;
    - UPDATE accounts SET balance=balance+1000 WHERE id = 2;
- **RDBMS transaction have ACID properties.**
    - Atomicity
        - All queries are executed as a single unit. If any query is failed, other queries are discarded.
- Consistency
    - When transaction is completed, all clients see the same data.
- Isolation
    - Multiple transactions (by same or multiple clients) are processed concurrently.
- Durable
    - When transaction is completed, all data is saved on disk.

- Transaction management
    - START TRANSACTION;
        Dml1
        Dml2
        Dml3
    - COMMIT WORK; → final save

    - START TRANSACTION;
        Dml1
        Dml2
        Dml3
    - ROLLBACK WORK; → Discard
- In MySQL autocommit variable is by default 1. So each DML command is auto-committed into database.
    - SELECT @@autocommit;
- Changing autocommit to 0, will create new transaction immediately after current transaction is completed. This setting can be made permanent in config file.
    - SET autocommit=0

Transactions
* Transaction is set of DML operations executed as a single unit.
    * If any operation from the set fails, the other operations will be discarded.
* START TRANSACTION;
    * If no transaction started, every DML operation is by default auto-commited.
    * Once transaction is started, all DML operation changes will be saved in temporary tables on server (not in main table). The temporary table internally created for each transaction.
    * During transaction when SELECT query is executed, the merged (original table + changes) result is sent to that client.
    * Transaction is completed when COMMIT or ROLLBACK is done.
* COMMIT WORK; -- WORK is optional ANSI keyword.
    * All changes recorded in the temporary table (of that tx) are permanently saved into main table.
* ROLLBACK WORK; -- WORK is optional ANSI keyword.
    * All changes recorded in the temporary table (of that tx) are permanently discarded.
* Any DML operation performed after completion of transaction (COMMIT or ROLLBACK) will be again auto-commited.

```SQL
-- banking: accounts table (accid, type, balance, ...)
-- transfer rs. 5000/- from account 1 to account 2.
  • UPDATE accounts SET balance = balance - 5000 WHERE accid = 1;
  • UPDATE accounts SET balance = balance + 5000 WHERE accid = 2;
```SQL

-- banking: accounts table (accid, type, balance, ...)
  • CREATE TABLE accounts(accid INT, type VARCHAR(20), balance DECIMAL(10,2));
  • INSERT INTO accounts VALUES (1, 'Saving', 50000);
  • INSERT INTO accounts VALUES (2, 'Saving', 500);
  • SELECT * FROM accounts; // output: 1=50000, 2=500

  • START TRANSACTION;
  • UPDATE accounts SET balance = balance - 5000 WHERE accid = 1;
  • SELECT * FROM accounts;  // output: 1=45000, 2=500

- `UPDATE accounts SET balance = balance + 5000 WHERE accid = 2;`
- `SELECT * FROM accounts; // output: 1=45000, 2=5500`
- `COMMIT WORK;`
- `SELECT * FROM accounts; //output: 1=45000, 2=5500`

- `START TRANSACTION;`
- `UPDATE accounts SET balance = balance - 2000 WHERE accid = 1;`
- `SELECT * FROM accounts; //output: 1=43000, 2=5500`

- `UPDATE accounts SET balance = balance + 2000 WHERE accid = 2;`
- `SELECT * FROM accounts; // output: 1=43000, 2=7500`
- `ROLLBACK WORK;`
- `SELECT * FROM accounts;  // output: 1=45000, 2=5500`

```

Transaction

- Save-point is state of database tables (data) at the moment (within a transaction).
- It is advised to create save-points at end of each logical section of work.
- Database user may choose to rollback to any of the save-point.
- Transaction management with Save-points
    - START TRANSACTION;
    - SAVEPOINT sa1;
    - SAVEPOINT sa2;
    - ROLLBACK TO sa1;
    - COMMIT; // or ROLLBACK
- Commit always commit the whole transaction.
- ROLLBACK or COMMIT clears all save-points
- Transaction is set of DML statements.
- If any DDL statement is executed, current transaction is automatically committed.
- Any power failure, system or network failure automatically rollback current state.
- Transactions are isolated from each other and are consistent

```
# Transactions
* autocommit=1 (default setting for MySQL), means there will be separate transaction for each DML query. This transaction is committed for each query.
* autocommit can be changed at session level or global level.
    * session level
```
```

        * applicable only for current client session (when client exit, setting is restored).
        * SET autocommit = 0;
    * global level
        * applicable for all sessions of all users.
        * set into my.ini file (by database admin).
* autocommit=0, means a transaction is started. On each commit/rollback, current transaction is completed and immediately new transaction is started (automatically).

```SQL
SELECT @@autocommit; //-- 1

SET autocommit = 0;

SELECT @@autocommit; //-- 0
```

- SELECT * FROM dept;
- DELETE FROM dept;
- SELECT * FROM dept;
- ROLLBACK;

```
-- current tx is completed and new tx is started here.
```

- SELECT * FROM dept;
- INSERT INTO dept VALUES(50, 'SECURITY', 'JAMMU');
- SELECT * FROM dept;
- ROLLBACK;
- SELECT * FROM dept;
- EXIT;

```SQL
SELECT @@autocommit;//-- 1
```

- START TRANSACTION;
- SELECT * FROM dept; //-- 10, 20, 30, 40
- INSERT INTO dept VALUES(50, 'SECURITY', 'JAMMU');
- SELECT * FROM dept; //-- 10, 20, 30, 40, 50

```
-- DDL command -- commit the current tx i.e. all changes are saved permanently.
-- since commit is done current, tx is completed.
   • CREATE TABLE temp(id INT, val CHAR(20));

-- nothing to discard -- changes were already saved.
   • ROLLBACK;

   • SELECT * FROM dept; //-- 10, 20, 30, 40, 50
```

* When EXIT is done, current tx is rollbacked.

```SQL
   • START TRANSACTION;
   • SELECT * FROM dept; // -- 10, 20, 30, 40, 50
   • DELETE FROM dept WHERE deptno=50;
   • SELECT * FROM dept; // -- 10, 20, 30, 40
   • EXIT;
```

```SQL
-- on new login
   • SELECT * FROM dept; //-- 10, 20, 30, 40, 50
```

Row locking

• When an user update or delete a row (within a transaction), that row is locked and becomes read -only for other users.
• The other users see old row values, until transaction is committed by first user.
• If other users try to modify or delete such locked row, their transaction processing is blocked until row is unlocked.
• Other users can INSERT into that table. Also they can UPDATE or DELETE other rows.
• The locks are automatically released when COMMIT/ROLLBACK is done by the user.
• This whole process is done automatically in MySQL. It is called as "OPTIMISTIC LOCKING"
• Manually locking the row in advanced before issuing UPDATE or DELETE is known as "PESSIMISTIC LOCKING".
• This is done by appending FOR UPDATE to the SELECT query.
• It will lock all selected rows, until transaction is committed or rollbacked.

• If these rows are already locked by another users, the SELECT operation is blocked until rows lock is released.
• By default MySQL does table locking. Row locking is possible only when table is indexed on the column

Optimistic Locking
* Automatically done by MySQL after update/delete row within transaction.

Pessimistic Locking
* User can lock a row in advance, before update/delete that row (within transaction).

```SQL
-- edac user
-- (1)
  • START TRANSACTION;

-- (2)
  • SELECT * FROM dept WHERE deptno=40 FOR UPDATE;
-- lock the row for update/delete

-- (4)
  • UPDATE dept SET loc='BOSTON' WHERE deptno=40;

-- (5)
  • COMMIT;
-- lock released
```
```SQL
-- root user
-- (3)
  • SELECT * FROM dept WHERE deptno=40 FOR UPDATE;
-- blocked

-- (6)
-- row is locked for this user transaction.
```

Entity Relations

• To avoid redundancy of the data, data should be organized into multiple tables so that tables are related to each other.

• The relations can be one of the following

- One to One
- One to Many
- Many to One
- Many to Many

• Entity relations is outcome of Normalization process.

SQL Joins

• Join statements are used to SELECT data from multiple tables using single query.

• Typical RDBMS supports following types of joins:

• Cross Join • Inner Join • Left Outer Join • Right Outer Join • Full Outer Join • Self join

SQL Joins

```SQL
- DROP TABLE IF EXISTS depts;
- DROP TABLE IF EXISTS emps;
- DROP TABLE IF EXISTS addr;
- DROP TABLE IF EXISTS meeting;
- DROP TABLE IF EXISTS emp_meeting;

- CREATE TABLE depts (deptno INT, dname VARCHAR(20));
- INSERT INTO depts VALUES (10, 'DEV');
- INSERT INTO depts VALUES (20, 'QA');
- INSERT INTO depts VALUES (30, 'OPS');
- INSERT INTO depts VALUES (40, 'ACC');

- CREATE TABLE emps (empno INT, ename VARCHAR(20), deptno INT, mgr INT);
- INSERT INTO emps VALUES (1, 'Amit', 10, 4);
- INSERT INTO emps VALUES (2, 'Rahul', 10, 3);
- INSERT INTO emps VALUES (3, 'Nilesh', 20, 4);
- INSERT INTO emps VALUES (4, 'Nitin', 50, 5);
- INSERT INTO emps VALUES (5, 'Sarang', 50, NULL);

- CREATE TABLE addr(empno INT, tal VARCHAR(20), dist VARCHAR(20));
- INSERT INTO addr VALUES (1, 'Gad', 'Kolhapur');
```

```
- INSERT INTO addr VALUES (2, 'Karad', 'Satara');
- INSERT INTO addr VALUES (3, 'Junnar', 'Pune');
- INSERT INTO addr VALUES (4, 'Wai', 'Satara');
- INSERT INTO addr VALUES (5, 'Karad', 'Satara');

- CREATE TABLE meeting (meetno INT, topic VARCHAR(20), venue
  VARCHAR(20));
- INSERT INTO meeting VALUES (100, 'Scheduling', 'Director Cabin');
- INSERT INTO meeting VALUES (200, 'Annual meet', 'Board Room');
- INSERT INTO meeting VALUES (300, 'App Design', 'Co-director
  Cabin');

- CREATE TABLE emp_meeting (meetno INT, empno INT);
- INSERT INTO emp_meeting VALUES (100, 3);
- INSERT INTO emp_meeting VALUES (100, 4);
- INSERT INTO emp_meeting VALUES (200, 1);
- INSERT INTO emp_meeting VALUES (200, 2);
- INSERT INTO emp_meeting VALUES (200, 3);
- INSERT INTO emp_meeting VALUES (200, 4);
- INSERT INTO emp_meeting VALUES (200, 5);
- INSERT INTO emp_meeting VALUES (300, 1);
- INSERT INTO emp_meeting VALUES (300, 2);
- INSERT INTO emp_meeting VALUES (300, 4);
```

Cross Join

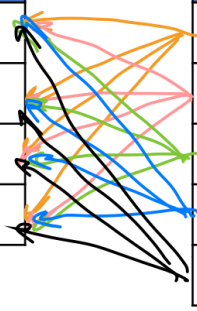
• Compares each row of Table1 with every row of Table2.

• Yields all possible combinations of Table1 and Table2.

• In MySQL, The larger table is referred as "Driving Table", while smaller table is referred as "Driven Table". Each row of Driving table is combined with every row of Driven table.

• Cross join is the fastest join, because there is no condition check involved

deptno	dname
10	DEV
20	QA
30	OPS
40	ACC



empno	ename	deptno
1	Amit	10
2	Rahul	10
3	Nilesh	20
4	Nitin	50
5	Sarang	50

Cross Join

```SQL
  • SELECT ename, dname FROM emps
    CROSS JOIN depts;

-- if column names in both tables are same,
-- to avoid conflicts, column names are prepended by tablename.
  • SELECT emps.ename, depts.dname FROM emps
    CROSS JOIN depts;

-- table alias is used to shorten the table names while selecting
columns
  • SELECT e.ename, d.dname FROM emps AS e
    CROSS JOIN depts AS d;

-- AS keyword optional
  • SELECT e.ename, d.dname FROM emps e
    CROSS JOIN depts d;

  • SELECT e.ename, d.dname FROM depts d
    CROSS JOIN emps e;
```

Inner Join

• The inner JOIN is used to return rows from both tables that satisfy the join condition.
• Non-matching rows from both tables are skipped.
• If join condition contains equality check, it is referred as equi-join; otherwise it is non-equi-join.

```
  • SELECT e.ename, d.dname FROM depts d
    INNER JOIN emps e ON e.deptno = d.deptno;
```

deptno	dname
10	DEV
20	QA
30	OPS
40	ACC

depts

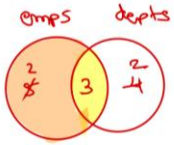
empno	ename	deptno
1	Amit	10
2	Rahul	10
3	Nilesh	20
4	Nitin	50
5	Sarang	50

emps

Left Outer Join

deptno	dname
10	DEV
20	QA
30	OPS
40	ACC

empno	ename	deptno
1	Amit	10
2	Rahul	10
3	Nilesh	20
4	Nitin	50
5	Sarang	50




```
for each e in emps
{   found = false;
    for each d in depts
    {   if (e.deptno == d.deptno) {
            print (e.ename, d.dname);
            found = true;
        }
    }
    if (found == false)
        print (e.ename, NULL);
}
```

```
Select e.ename, d.dname from emps e
left outer join depts d on e.deptno = d.deptno;
```

- Left outer join is used to return matching rows from both tables along with additional rows in left table.
- Corresponding to additional rows in left table, right table values are taken as NULL.
- OUTER keyword is optional.

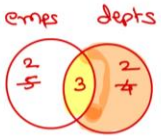
```SQL
-- intersection + extra emps
  • SELECT e.ename, d.dname FROM emps e
    LEFT OUTER JOIN depts d ON e.deptno = d.deptno;

-- intersection + extra depts
  • SELECT e.ename, d.dname FROM depts d
    LEFT OUTER JOIN emps e ON e.deptno = d.deptno;
```

Right Outer Join

deptno	dname
10	DEV
20	QA
30	OPS
40	ACC

empno	ename	deptno
1	Amit	10
2	Rahul	10
3	Nilesh	20
4	Nitin	50
5	Sarang	50




```
Select e.ename, d.dname from emps e
right outer join depts d on e.deptno = d.deptno;
```

- Right outer join is used to return matching rows from both tables along with additional rows in right table.
- Corresponding to additional rows in right table, left table values are taken as NULL.
- OUTER keyword is optional.

```SQL
-- intersection + extra depts
  • SELECT e.ename, d.dname FROM emps e
    RIGHT OUTER JOIN depts d ON e.deptno = d.deptno;
```

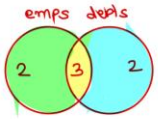
```
-- intersection + extra emps
   • SELECT e.ename, d.dname FROM depts d
     RIGHT OUTER JOIN emps e ON e.deptno = d.deptno;
```

Full Outer Join

* Not supported in MySQL RDBMS.
* Full Outer Join is supported in Oracle, MS-SQL, ...

deptno	dname
10	DEV
20	QA
30	OPS
40	ACC

empno	ename	deptno
1	Amit	10
2	Rahul	10
3	Nilesh	20
4	Nitin	50
5	Sarang	50



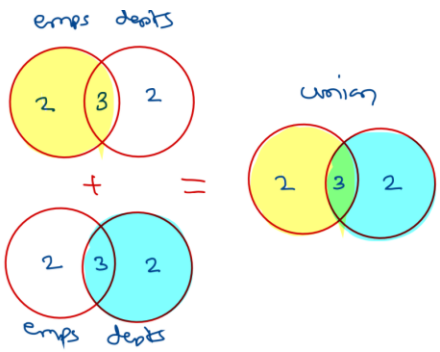
- Full join is used to return matching rows from both tables along with additional rows in both tables.
- Corresponding to additional rows in left or right table, opposite table values are taken as NULL.
- Full outer join is not supported in MySQL, but can be simulated using set operators.

```SQL
-- intersection + extra emps + extra depts
   • SELECT e.ename, d.dname FROM emps e
     FULL OUTER JOIN depts d ON e.deptno = d.deptno;
-- error in MySQL
```

Set Operators

ename	dname
Amit	DEV
Rahul	DEV
Nilesh	QA
NULL	OPS
NULL	ACC

ename	dname
Amit	DEV
Rahul	DEV
Nilesh	QA
Nitin	NULL
Sarang	NULL



- UNION operator is used to combine results of two queries. The common data is taken only once. It can be used to simulate full outer join.
- UNION ALL operator is used to combine results of two queries. Common data is repeated.

```SQL
-- simulation of full join
-- intersection (only once) + extra emps + extra depts
  • (SELECT e.ename, d.dname FROM emps e
    LEFT OUTER JOIN depts d ON e.deptno = d.deptno)
    UNION
  • (SELECT e.ename, d.dname FROM emps e
    RIGHT OUTER JOIN depts d ON e.deptno = d.deptno);

-- intersection (twice) + extra emps + extra depts
-- output of second query is appended to output of first query
  • (SELECT e.ename, d.dname FROM emps e
    LEFT OUTER JOIN depts d ON e.deptno = d.deptno)
    UNION ALL
  • (SELECT e.ename, d.dname FROM emps e
    RIGHT OUTER JOIN depts d ON e.deptno = d.deptno);
```

Self Join

- When join is done on same table, then it is known as "Self Join". The both columns in condition belong to the same table.
- Self join may be an inner join or outer join.

empno	ename	deptno	mgr
1	Amit	10	4
2	Rahul	10	3
3	Nilesh	20	4
4	Nitin	50	5
5	Sarang	50	NULL

empno	ename	deptno	mgr
1	Amit	10	4
2	Rahul	10	3
3	Nilesh	20	4
4	Nitin	50	5
5	Sarang	50	NULL

emps e

Amit - Nitin
Rahul - Nilesh
Nilesh - Nitin
Nitin - Sarang

mgrs m

Sarang
Nitin
Nilesh
Amit
Rahul

```
foreach e in emps
{
    foreach m in emps
    {
        if(e.mgr == m.empno)
            print(e.ename, m.ename)
    }
}
```

```
select e.ename, m.ename from emps e
inner join emps m on e.mgr = m.empno;
```

```SQL
-- self join -- inner join with same table -- intersection
  • SELECT e.ename, m.ename AS mname FROM emps e
    INNER JOIN emps m ON e.mgr = m.empno;

-- self join -- outer join -- intersection + extra from left table
(emps e)
  • SELECT e.ename, m.ename AS mname FROM emps e
    LEFT OUTER JOIN emps m ON e.mgr = m.empno;
```

Joins with other clauses

```SQL
- SELECT * FROM emps;
- SELECT * FROM depts;
- SELECT * FROM addr;

-- print ename, dname.
- SELECT e.ename, d.dname FROM emps e
  INNER JOIN depts d ON e.deptno = d.deptno;

-- print ename, tal of emp.
- SELECT e.ename, a.tal FROM emps e
  INNER JOIN addr a ON e.empno = a.empno;

-- print ename, dname and tal of emp.
- SELECT e.ename, d.dname, a.tal FROM emps e
  INNER JOIN depts d ON e.deptno = d.deptno
  INNER JOIN addr a ON e.empno = a.empno;

-- print ename, dname and tal of emp.
- SELECT e.ename, d.dname, a.tal FROM emps e
  LEFT OUTER JOIN depts d ON e.deptno = d.deptno
  INNER JOIN addr a ON e.empno = a.empno;
```

```SQL
- SELECT * FROM emps;
- SELECT * FROM depts;
- SELECT deptno, COUNT(empno) FROM emps
  GROUP BY deptno;

- SELECT e.ename, d.dname FROM emps e
  LEFT OUTER JOIN depts d ON e.deptno = d.deptno;

-- print dname and number of emps in dept.
- SELECT d.dname, COUNT(e.empno) FROM emps e
  LEFT OUTER JOIN depts d ON e.deptno = d.deptno
  GROUP BY d.dname;
```

```SQL
-- print ename, dname for emps with empno 2 and 3.
  • SELECT e.ename, d.dname FROM emps e
    INNER JOIN depts d ON e.deptno = d.deptno
    WHERE e.empno IN (2,3);
```

Joins

Examples

```SQL
USE sales;

-- Write a query that lists each order number followed by the name of the customer who made the order.
  • SELECT o.onum, c.cname FROM orders o
    INNER JOIN customers c ON o.cnum = c.cnum;

-- Write a query that gives the names of both the salesperson and the customer for each order along with the order number.
  • SELECT o.onum, s.sname, c.cname FROM orders o
    INNER JOIN customers c ON o.cnum = c.cnum
    INNER JOIN salespeople s ON o.snum = s.snum;

-- another syntax of writing join -- only support inner join -- not standard way of writing join.
  • SELECT o.onum, s.sname, c.cname FROM orders o, customers c,
    salespeople s
    WHERE o.cnum = c.cnum AND o.snum = s.snum;

-- Write a query that produces all customers serviced by salespeople with a commission above 12%. Output the customer's name, the salesperson's name, and the salesperson's rate of commission.
  • SELECT c.cname, s.sname, s.comm FROM customers c
    INNER JOIN salespeople s ON c.snum = s.snum
    WHERE s.comm > 0.12;

-- Write a query that calculates the amount of the salesperson's commission on each order by a customer with a rating above 100.
```

- SELECT o.onum, c.rating, o.amt, s.comm, o.amt * s.comm FROM orders o
  INNER JOIN customers c ON o.cnum = c.cnum
  INNER JOIN salespeople s ON o.snum = s.snum
  WHERE c.rating > 100;

-- Write a query that produces all pairs of salespeople who are living in the same city. Exclude combinations of salespeople with themselves as well as duplicate rows with the order reversed.

- SELECT * FROM salespeople;

- SELECT s1.sname, s2.sname, s1.city FROM salespeople s1
  INNER JOIN salespeople s2 ON s1.city = s2.city;

- SELECT s1.snum, s1.sname, s2.snum, s2.sname, s1.city FROM salespeople s1
  INNER JOIN salespeople s2 ON s1.city = s2.city
  WHERE s1.snum != s2.snum;

- SELECT s1.snum, s1.sname, s2.snum, s2.sname, s1.city FROM salespeople s1
  INNER JOIN salespeople s2 ON s1.city = s2.city
  WHERE s1.snum < s2.snum;

```

* Tables
```

STUDENTS

std	roll	name	address
1	1	A	Pune
1	2	B	Pune
1	3	C	Pune
2	1	D	Pune
2	2	E	Pune

MARKS

std	roll	subject	marks
1	1	Hin	20
1	1	Eng	25
1	1	Math	30
1	2	Hin	10
1	2	Eng	20
1	2	Math	22
2	1	Hin	8
2	1	Eng	20

```SQL
  • SELECT s.name, m.subject, m.marks FROM STUDENTS s
    INNER JOIN MARKS m ON s.std = m.std AND s.roll = m.roll;
```

DDL ± ALTER statement

- ALTER statement is used to do modification into table, view , function ,procedure.
- ALTER TABLE is used to change table structure.
- Add new column(s) into the table.
    - ALTER TABLE table ADD col TYPE;
    - ALTER TABLE table ADD c1 TYPE, c2 TYPE;
- Modify column of the table.
    - ALTER TABLE table MODIFY col NEW_TYPE;
- Rename column.
    - ALTER TABLE CHANGE old_col new_col TYPE;
- Drop a column
    - ALTER TABLE DROP COLUMN col;
- Rename table
    - ALTER TABLE table RENAME TO new_table;

ALTER TABLE

```SQL
  • USE dacdb;
  • SHOW TABLES;

  • CREATE TABLE temp (id INT, val CHAR(10));
  • DESC temp;
  • INSERT INTO temp VALUES (1, 'ABCD'), (2, 'WXYZ');
  • SELECT * FROM temp;

-- add new column
  • ALTER TABLE temp ADD sal DOUBLE;
  • DESC temp;
  • SELECT * FROM temp;
  • INSERT INTO temp VALUES (3, 'abc', 4550.0), (4, 'xyz', 6344.2);
  • SELECT * FROM temp;
```

```
-- modify data type
   • ALTER TABLE temp MODIFY sal DECIMAL(7,2);
   • DESC temp;
   • SELECT * FROM temp;

-- modify data type
   • ALTER TABLE temp MODIFY sal DECIMAL(3,2);
-- if data is not convertable to new type, ALTER TABLE fails.

-- rename column
   • ALTER TABLE temp CHANGE sal income DECIMAL(7,2);
   • DESC temp;
   • SELECT * FROM temp;

-- rename table
   • ALTER TABLE temp RENAME TO timepass;
   • DESC timepass;
   • SELECT * FROM timepass;

-- drop column
   • ALTER TABLE timepass DROP COLUMN val;
   • DESC timepass;
   • SELECT * FROM timepass;
```

Query performance

- Few RDBMS features ensure better query performance.
    - Index speed up execution of SELECT queries (search operations).
    - Correlated sub-queries execute faster.
- Query performance can observed using EXPLAIN statement.
    - EXPLAIN FORMAT=JSON SELECT «;
- EXPLAIN statement shows
    - Query cost (Lower is the cost, faster is the query execution).
    - Execution plan (Algorithm used to execute query e.g. loop, semi-join, materialization, etc).
- Optimizations can be enabled or disabled by optimizer_switch system variable.
    - SELECT @@optimizer_switch;
    - SET @@optimizer_switch=”materialization=off”;

Query Performance

* Query cost depends on
    * Machine settings
    * MySQL version
    * "Optimization settings"
* Lower query cost, means more efficient query.

```SQL
```

- SELECT job, SUM(sal) FROM emp
  WHERE job IN ('ANALYST', 'SALESMAN')
  GROUP BY job;

- SELECT job, SUM(sal) FROM emp
  GROUP BY job
  HAVING job IN ('ANALYST', 'SALESMAN');

- EXPLAIN FORMAT=JSON
- SELECT job, SUM(sal) FROM emp
  WHERE job IN ('ANALYST', 'SALESMAN')
  GROUP BY job;

- EXPLAIN FORMAT=JSON
- SELECT job, SUM(sal) FROM emp
  GROUP BY job
  HAVING job IN ('ANALYST', 'SALESMAN');

- SELECT e.ename, m.ename mname FROM emp e
  INNER JOIN emp m ON e.mgr = m.empno;

- EXPLAIN FORMAT=JSON
- SELECT e.ename, m.ename mname FROM emp e
  INNER JOIN emp m ON e.mgr = m.empno;

- SELECT e.ename, d.dname FROM emp e
  INNER JOIN dept d ON e.deptno = d.deptno;

- EXPLAIN FORMAT=JSON
- SELECT e.ename, d.dname FROM emp e
  INNER JOIN dept d ON e.deptno = d.deptno;

Index

- Index enable faster searching in tables by indexed columns.
- CREATE INDEX idx_name ON table(column);
- One table can have multiple indexes on different columns/order.
- Typically indexes are stored as some data structure (like BTREE or HASH) on disk.
- Indexes are updated during DML operations. So DML operation are slower on indexed tables.
- Index can be ASC or DESC.
    - It cause storage of key values in respective order (MySQL 8.x onwards).
    - ASC/DESC index is used by optimizer on ORDER BY queries.
- There are four types of indexes:
    - Simple index
        - CREATE INDEX idx_name ON table(column [ASC|DESC]);
- Unique index
    - CREATE UNIQUE INDEX idx_name ON table(column [ASC|DESC]);
    - Doesn't allow duplicate values.
- Composite index
    - CREATE INDEX idx_name ON table(column1 [ASC|DESC], column2 [ASC|DESC]);
    - Composite index can also be unique. Do not allow duplicate combination of columns.
- Clustered index
    - PRIMARY index automatically created on Primary key for row lookup.
    - If primary key is not available, hidden index is created on synthetic column.
    - It is maintained in tabular form and its reference is used in other indexes
- Indexes should be created on shorter (INT,CHAR) columns to save disk space.
- Few RDBMS do not allow indexes on external columns i.e. TEXT, BLOB.
- MySQL support indexing on TEXT/BLOB up to n characters.
    - CREATE TABLE test (blob_col BLOB, «, INDEX(blob_col(10)));
- To list all indexes on table:
    - SHOW INDEXES ON table;
- To drop an index:
    - DROP INDEX idx_name ON table;
- When table is dropped, all indexes are automatically dropped.
- Indexes should not be created on the columns not used frequent search, ordering or grouping operations.
- Columns in join operation should be indexed for better performance

Indexes
* syntax: CREATE INDEX idx_name ON tablename(colname);

```SQL
  - SELECT * FROM emp WHERE job='CLERK';

EXPLAIN FORMAT=JSON
  - SELECT * FROM emp WHERE job='CLERK'; //-- cost = 1.65
    DESC emp;
    CREATE INDEX idx_emp_job ON emp(job);
    DESC emp;

  - EXPLAIN FORMAT=JSON
  - SELECT * FROM emp WHERE job='CLERK';//- cost = 0.90
  - SELECT * FROM emp WHERE deptno=20;

  - EXPLAIN FORMAT=JSON
  - SELECT * FROM emp WHERE deptno=20;//-- cost = 1.65
    CREATE INDEX idx_emp_deptno ON emp(deptno);

  - EXPLAIN FORMAT=JSON
  - SELECT * FROM emp WHERE deptno=20; //-- cost = 1.00
```

Indexes

```SQL
    - DESC emp;
--index is sorted in desc order of hire
    - CREATE INDEX idx_emp_hire ON emp(hire DESC);

    - SELECT * FROM emp ORDER BY hire DESC;
-- index is sorted in asc order of mgr
    - CREATE INDEX idx_emp_mgr ON emp(mgr ASC);

    - DESC emp;
```

Types of indexes

```SQL
-- Regular Index
CREATE INDEX idx_emp_sal ON emp(sal ASC);

SELECT * FROM emp WHERE sal = 3000.0;

-- Unique Index
CREATE UNIQUE INDEX idx_emp_ename ON emp(ename ASC);

DESC emp;

SELECT * FROM emp WHERE ename='KING';

INSERT INTO emp(empno,ename,sal) VALUES(1001, 'KING', 6000.0);
-- error: duplicate values not allowed - due to UNIQUE index.

-- Composite Index -- on multiple columns
CREATE INDEX idx_emp_deptno_job ON emp(deptno, job);

SELECT * FROM emp WHERE deptno=20 AND job='CLERK';

DROP TABLE IF EXISTS students;

CREATE TABLE students (std INT, roll INT, name VARCHAR(20), addr
VARCHAR(50));

INSERT INTO students VALUES (1,1,'A','Pune'), (1,2,'B','Pune'),
(1,3,'C','Pune'), (2,1,'D','Pune'), (2,2,'E','Pune');

SELECT * FROM students;

-- Composite UNIQUE index -- Duplicate combination (std,roll) is not
allowed.
CREATE UNIQUE INDEX idx_stud ON students(std,roll);

DESC students;

INSERT INTO students VALUES (3, 6, 'F', 'Pune');
```

```
INSERT INTO students VALUES (3, 6, 'G', 'Pune');
-- error: student with std=3, roll=6 already exists.

SELECT * FROM students WHERE std=3 AND roll=6;

-- clustered index
ALTER TABLE emp ADD PRIMARY KEY(empno);

DESC emp;
```

```SQL
SHOW INDEXES FROM emp;

DROP INDEX idx_emp_deptno_job ON emp;

SELECT * FROM emp WHERE deptno=20 AND job='CLERK';
-- slow down

SHOW INDEXES FROM emp;
```

Constraints

- Constraints are restrictions imposed on columns.
- There are five constraints
    - NOT NULL
    - UNIQUE
    - PRIMARY KEY
    - FOREIGN KEY
    - CHECK
- Few constraints can be applied at either column level or table level. Few constraints can be applied on both.
- Optionally constraint names can be mentioned while creating the constraint. If not given, it is auto-generated.
- Each DML operation check the constraints before manipulating the values. If any constraint is violated, error is raised.

- **NOT NULL**
    - NULL values are not allowed.
    - Can be applied at column level only.
    - CREATE TABLE WDEOH(F1 TYPE NOT NULL, «);

NOT NULL

```SQL
- CREATE TABLE t1(c1 INT NOT NULL, c2 CHAR(20) NOT NULL, c3 DECIMAL(5,2));

- DESC t1;

- INSERT INTO t1 VALUES(1, 'A', 23.4);

- INSERT INTO t1 VALUES(1, NULL, 23.4);
  --error

- INSERT INTO t1 VALUES(NULL, NULL, 23.4);
  --error

- INSERT INTO t1 (c1,c3) VALUES(2, 12.4);
  --error

- INSERT INTO t1 VALUES(3, 'C', NULL);
  --allowed
```

- **UNIQUE**
    - Duplicate values are not allowed.
    - NULL values are allowed.
    - Not applicable for TEXT and BLOB.
    - UNIQUE can be applied on one or more columns.
    - Internally creates unique index on the column (fast searching).
    - Can be applied at column level or table level.
        - CREATE TABLE table(c1 TYPE UNIQUE, «);
        - CREATE TABLE table(c1 TYPE, «, UNIQUE(F1));
        - CREATE TABLE WDEOH(F1 TYPE, «, CONSTRAINT constraint_name UNIQUE(c1));

UNIQUE

```SQL
- CREATE TABLE people(
  id INT UNIQUE NOT NULL, -- column level constraint
  name VARCHAR(20),
  addr VARCHAR(80),
  email CHAR(40),
  mobile CHAR(16),
  CONSTRAINT c1 UNIQUE(email), -- table level named constraint
  UNIQUE(mobile), -- table level constraint
  UNIQUE(name,addr) -- must be table level constraint
  );

- DESC people;

- SHOW INDEXES FROM people;

- INSERT INTO people VALUES (1, 'A', 'Pune', 'a@gmail.com',
  '1234567890');

- INSERT INTO people VALUES (2, 'B', 'Pune', NULL, NULL);

- INSERT INTO people VALUES (3, 'C', 'Pune', 'c@gmail.com', NULL);

- INSERT INTO people VALUES (4, 'D', 'Pune', 'd@gmail.com',
  '1234567890');
-- mobile UNIQUE index -- duplicate not allowed

- INSERT INTO people VALUES (NULL, 'E', 'Pune', 'e@gmail.com',
  NULL);
-- id is NOT NULL

- INSERT INTO people VALUES (5, 'C', 'Pune', 'f@gmail.com', NULL);
-- error: name & addr compbination cannot be duplicated.

- SHOW CREATE TABLE people;
```

- **PRIMARY KEY**
    - Column or set of columns that uniquely identifies a row.
    - Only one primary key is allowed for a table.
    - Primary key column cannot have duplicate or NULL values.
    - Internally index is created on PK column.
    - TEXT/BLOB cannot be primary key.
    - If no obvious choice available for PK, composite or surrogate PK can be created.
    - Creating PK for a table is a good practice.
    - PK can be created at table level or column level.
    - CREATE TABLE WDEOH(F1 TYPE PRIMARY KEY, «);
    - CREATE TABLE table(c1 TYPE, «, PRIMARY KEY(c1));
    - CREATE TABLE WDEOH(F1 TYPE, «, CONSTRAINT constraint_name PRIMARY KEY(c1));
    - CREATE TABLE table(c1 TYPE, c2 TYPE, «, PRIMARY KEY(c1, c2));

PRIMARY KEY

* Primary Key = Unique + Not NULL
    * Cannot be duplicated
    * Compulsory
* Unique columns can be multiple, but Primary Key column is only one per table.
* Primary Key identifies each record in the table.

```SQL
- DROP TABLE IF EXISTS customers;

- CREATE TABLE customers(
  email CHAR(40) PRIMARY KEY, -- column level
  name VARCHAR(40),
  addr VARCHAR(100),
  age INT
  );

-- OR

- CREATE TABLE customers(
  email CHAR(40),
  name VARCHAR(40),
  addr VARCHAR(100),
```

```
    age INT,
    PRIMARY KEY (email) -- table level
    );

-- OR
```

- `CREATE TABLE customers(`
  `email CHAR(40),`
  `name VARCHAR(40),`
  `addr VARCHAR(100),`
  `age INT,`
  `CONSTRAINT pk_customers PRIMARY KEY (email) -- table level named`
  `);`

- `DESC customers;`

- `INSERT INTO customers VALUES ('a@gmail.com', 'A', 'Pune', 23);`
- `INSERT INTO customers VALUES ('b@gmail.com', 'B', 'Pune', 24);`
- `INSERT INTO customers VALUES (NULL, 'C', 'Pune', 22);`

```
-- error: cannot be null
```

- `INSERT INTO customers VALUES ('b@gmail.com', 'D', 'Pune', 20);`

```
-- error: cannot be duplicated
```

- `SHOW INDEXES FROM customers;`

```
-- composite primary key
```

- `DROP TABLE IF EXISTS students;`
- `CREATE TABLE students (`
  `std INT,`
  `roll INT,`
  `name VARCHAR(20),`
  `addr VARCHAR(50),`
  `PRIMARY KEY(std,roll)`
  `);`
- `DESC students;`

```
-- surrogate primary key
```

- `CREATE TABLE persons(`
  `id INT PRIMARY KEY AUTO_INCREMENT,`
  `name CHAR(20),`
  `age INT`
  `);`

- `INSERT INTO persons(name,age) VALUES('A', 22);`
- `INSERT INTO persons(name,age) VALUES('B', 23);`
- `INSERT INTO persons(name,age) VALUES('C', 25);`
- `INSERT INTO persons(name,age) VALUES('D', 21);`
- `INSERT INTO persons(name,age) VALUES('E', 20);`

- `SELECT * FROM persons;`

- `ALTER TABLE persons AUTO_INCREMENT=1000;`
- `INSERT INTO persons(name,age) VALUES('F', 19);`
- `INSERT INTO persons(name,age) VALUES('G', 18);`

- `SELECT * FROM persons;`

```
```

• FOREIGN KEY

- Column or set of columns that references a column of some table.
- If column belongs to the same table, it is "self referencing".
- Foreign key constraint is specified on child table column.
- FK can have duplicate values as well as null values.
- FK constraint is applied on column of child table (not on parent table).
- Child rows cannot be deleted, until parent rows are deleted.
- MySQL have ON DELETE CASCADE clause to ensure that child rows are automatically deleted, when parent row is deleted. ON UPDATE CASCADE clause does same for UPDATE operation.
- By default foreign key checks are enabled. They can be disabled by
  - SET @@foreign_key_checks = 0;
- FK constraint can be applied on table level as well as column level.
- CREATE TABLE FKLOG(F1 TYPE, «, FOREIGN KEY (F1) REFERENCES parent(col))

Foreign Key

```SQL
- DROP TABLE IF EXISTS depts;
- DROP TABLE IF EXISTS emps;

- CREATE TABLE depts (deptno INT, dname VARCHAR(20), PRIMARY KEY(deptno));
- INSERT INTO depts VALUES (10, 'DEV');
- INSERT INTO depts VALUES (20, 'QA');
- INSERT INTO depts VALUES (30, 'OPS');
- INSERT INTO depts VALUES (40, 'ACC');


- CREATE TABLE emps (
  empno INT,
  ename VARCHAR(20),
  deptno INT,
  mgr INT,
  FOREIGN KEY (deptno) REFERENCES depts(deptno)
  );
- DESC emps;

- INSERT INTO emps VALUES (1, 'Amit', 10, 4);
- INSERT INTO emps VALUES (2, 'Rahul', 10, 3);
- INSERT INTO emps VALUES (3, 'Nilesh', 20, 4);
- INSERT INTO emps VALUES (4, 'Nitin', 50, 5);
-- error: deptno=50 dept does not exists.
- INSERT INTO emps VALUES (5, 'Sarang', 50, NULL);
-- error: deptno=50 dept does not exists.

- DROP TABLE IF EXISTS emps;


- CREATE TABLE emps (
  empno INT PRIMARY KEY,
  ename VARCHAR(20),
  deptno INT,
  mgr INT,
  FOREIGN KEY (mgr) REFERENCES emps(empno),
```

```
    FOREIGN KEY (deptno) REFERENCES depts(deptno)
    );
  • DESC emps;

  • INSERT INTO emps VALUES (5, 'Sarang', NULL, NULL);
  • INSERT INTO emps VALUES (4, 'Nitin', NULL, 5);
--fk can be NULL
  • INSERT INTO emps VALUES (1, 'Amit', 10, 4);
  • INSERT INTO emps VALUES (3, 'Nilesh', 20, 4);
  • INSERT INTO emps VALUES (2, 'Rahul', 10, 3);
```

```SQL
-- alternate FOREIGN key syntax
-- doesn't work well in MySQL.

  • CREATE TABLE emps (
    empno INT PRIMARY KEY,
    ename VARCHAR(20),
    deptno INT REFERENCES depts(deptno),
    mgr INT,
    );

  • DESC emps;
```

```SQL
-- FK to composite PK.
  • CREATE TABLE marks(
    std INT,
    roll INT,
    subject CHAR(20),
    marks INT DEFAULT 0.0,
    FOREIGN KEY (std,roll) REFERENCES students(std,roll)
    );

  • INSERT INTO marks(std,roll,subject) VALUES (1,1,'Hin');
```

Foreign Key Constraints

```SQL
- DROP TABLE IF EXISTS depts;
- DROP TABLE IF EXISTS emps;

- CREATE TABLE depts (deptno INT, dname VARCHAR(20), PRIMARY KEY(deptno));
- INSERT INTO depts VALUES (10, 'DEV');
- INSERT INTO depts VALUES (20, 'QA');
- INSERT INTO depts VALUES (30, 'OPS');
- INSERT INTO depts VALUES (40, 'ACC');

- CREATE TABLE emps (
  empno INT PRIMARY KEY,
  ename VARCHAR(20),
  deptno INT,
  mgr INT,
  FOREIGN KEY (mgr) REFERENCES emps(empno),
  FOREIGN KEY (deptno) REFERENCES depts(deptno)
  );
- DESC emps;

- INSERT INTO emps VALUES (5, 'Sarang', NULL, NULL);
- INSERT INTO emps VALUES (4, 'Nitin', NULL, 5);
- INSERT INTO emps VALUES (1, 'Amit', 10, 4);
- INSERT INTO emps VALUES (3, 'Nilesh', 20, 4);
- INSERT INTO emps VALUES (2, 'Rahul', 10, 3);
```

```SQL
- SHOW CREATE TABLE depts;
-- primary key = deptno

- SHOW CREATE TABLE emps;
-- Foreign key = deptno

- SELECT * FROM depts;

- SELECT * FROM emps;
```

```
- DELETE FROM depts WHERE deptno=10;
-- error: cannot delete parent row.

- UPDATE depts SET deptno=60 WHERE deptno=10;
-- error: cannot update parent row's primary key.

- DELETE FROM depts WHERE deptno=40;
-- okay: no child row for deptno=40.
```

```SQL
- DROP TABLE IF EXISTS emps;
- DROP TABLE IF EXISTS depts;

- CREATE TABLE depts (deptno INT, dname VARCHAR(20), PRIMARY
  KEY(deptno));
- INSERT INTO depts VALUES (10, 'DEV');
- INSERT INTO depts VALUES (20, 'QA');
- INSERT INTO depts VALUES (30, 'OPS');
- INSERT INTO depts VALUES (40, 'ACC');

- CREATE TABLE emps (
  empno INT PRIMARY KEY,
  ename VARCHAR(20),
  deptno INT,
  mgr INT,
  FOREIGN KEY (deptno) REFERENCES depts(deptno) ON DELETE CASCADE
  ON UPDATE CASCADE
  );
- DESC emps;

- INSERT INTO emps VALUES (5, 'Sarang', NULL, NULL);
- INSERT INTO emps VALUES (4, 'Nitin', NULL, 5);
- INSERT INTO emps VALUES (1, 'Amit', 10, 4);
- INSERT INTO emps VALUES (3, 'Nilesh', 20, 4);
- INSERT INTO emps VALUES (2, 'Rahul', 10, 3);

- SELECT * FROM depts;
- SELECT * FROM emps;
```

```
  • UPDATE depts SET deptno=60 WHERE deptno=10;
-- update deptno=10 to deptno=60 in depts table (parent row)
-- also update deptno=60 in emps table (child rows -- Amit & Rahul)

  • SELECT * FROM depts;

  • SELECT * FROM emps;

  • DELETE FROM depts WHERE deptno=60;
-- delete deptno=60 from depts table (parent row)
-- also delete deptno=60 in emps table (child rows -- Amit & Rahul)

  • SELECT * FROM depts;
  • SELECT * FROM emps;
```

```SQL
SELECT @@foreign_key_checks;

SET @@foreign_key_checks = 0;

SELECT @@foreign_key_checks;

INSERT INTO emps VALUES(6, 'Vishal', 100, 3);
-- not good practice
-- allowed: because foreign_key_checks are disabled.

SELECT * FROM emps;

SELECT * FROM depts;

SET @@foreign_key_checks = 1;

INSERT INTO emps VALUES(7, 'Smita', 100, 3);
-- error: because foreign_key_checks are enabled.

SELECT * FROM emps;
```

- **CHECK**
  - CHECK is integrity constraint in SQL.
  - CHECK constraint specifies condition on column.
  - Data can be inserted/updated only if condition is true; otherwise error is raised.
  - CHECK constraint can be applied at table level or column level.
  - F- CREATE TABLE tabke(c1 TYPE, c2 TYPE CHECK condition1, «, CHECK condition2);

CHECK constraint

```SQL
  • SELECT @@version;
-- must be >= 8.0.16

  • CREATE TABLE newemp(
    id INT PRIMARY KEY,
    name CHAR(20) NOT NULL,
    sal DOUBLE CHECK (sal >= 5000),
    comm DOUBLE,
    job CHAR(20) CHECK (job IN ('CLERK', 'MANAGER', 'SALESMAN')),
    age INT CHECK (age >= 18),
    CHECK (sal + comm <= 50000)
    );

  • INSERT INTO newemp VALUES (1, 'A', 10000, 1000, 'MANAGER', 20);

  • INSERT INTO newemp VALUES (2, 'B', 4000, 1000, 'MANAGER', 20);
-- error: sal < 5000

  • INSERT INTO newemp VALUES (3, 'C', 40000, 11000, 'MANAGER', 20);
-- error: sal + comm > 50000

  • INSERT INTO newemp VALUES (4, 'D', 8000, 2000, 'MANAGER', 16);
-- error: age < 18

  • INSERT INTO newemp VALUES (5, 'E', 8000, 2000, 'PRESIDENT', 40);
-- error: job is not valid
```

Sub queries

- Sub-query is query within query. Typically it work with SELECT statements.
- Output of inner query is used as input to outer query.
- If no optimization is enabled, for each row of outer query result, sub-query is executed once. This reduce performance of sub-query.
- Single row sub-query
    - Sub-query returns single row.
    - Usually it is compared in outer query using relational operators

- **Multi-row sub-query**
    - Sub-query returns multiple rows.
    - Usually it is compared in outer query using operators like IN, ANY or ALL.
        - IN operator checks for equality with results from sub-queries.
        - ANY operator compares with one of the result from sub-queries.
        - ALL operator compares with all the results from sub-queries.

- **Correlated sub-query**
- If number of results from sub-query are reduced, query performance will increase.
- This can be done by adding criteria (WHERE clause) in sub-query based on outer query row.
- Typically correlated sub-query use IN, ALL, ANY and EXISTS operators.


```SQL
# Sub-query

## Single Row Sub-queries

```SQL
-- find the employee with max sal.

SELECT * FROM emp WHERE sal = MAX(sal);
-- error: group fn cannot be used in where clause.

SELECT MAX(sal) FROM emp;
SELECT * FROM emp WHERE sal = 5000.0;
-- working, but manually copying result of first query into second query.
```
```

```SQL
SET @maxsal = (SELECT MAX(sal) FROM emp);
SELECT @maxsal;
SELECT * FROM emp WHERE sal = @maxsal;
-- working, but two queries are involved.

SELECT * FROM emp WHERE sal = (SELECT MAX(sal) FROM emp);
-- working in single query -- Sub-query approach.
```

```SQL
-- find all employees whose sal is more than average sal of all
employees.
SET @avgsal = (SELECT AVG(sal) FROM emp);
SELECT * FROM emp WHERE sal > @avgsal;
SELECT @avgsal;

SELECT * FROM emp WHERE sal > (SELECT AVG(sal) FROM emp);
```

```SQL
-- find employees with second highest salary.
SELECT * FROM emp ORDER BY sal DESC;

SELECT * FROM emp ORDER BY sal DESC LIMIT 1,1;
-- will not show all emps having 2nd highest sal

SELECT DISTINCT sal FROM emp ORDER BY sal DESC LIMIT 1,1;

SET @avg2 = (SELECT DISTINCT sal FROM emp ORDER BY sal DESC LIMIT
1,1);
SELECT @avg2; -- 3000.00
SELECT * FROM emp WHERE sal = @avg2;

SELECT * FROM emp WHERE sal = (SELECT DISTINCT sal FROM emp ORDER BY
sal DESC LIMIT 1,1);
```

```SQL
-- find employees with third highest salary.
```

```SQL
SELECT DISTINCT sal FROM emp ORDER BY sal DESC LIMIT 2,1;

SELECT * FROM emp WHERE sal = (SELECT DISTINCT sal FROM emp ORDER BY
sal DESC LIMIT 2,1);
```

```SQL
-- find employees with third lowest salary.
SELECT DISTINCT sal FROM emp ORDER BY sal ASC LIMIT 2,1;

SELECT * FROM emp WHERE sal = (SELECT DISTINCT sal FROM emp ORDER BY
sal ASC LIMIT 2,1);
```

Multi-Row sub-query

```SQL
SELECT sal FROM emp WHERE job='MANAGER';

-- Find all employees whose sal is more than any manager.
SELECT MIN(sal) FROM emp WHERE job='MANAGER';

SELECT * FROM emp WHERE sal > (SELECT MIN(sal) FROM emp WHERE
job='MANAGER');
-- solution using single row sub-query

SELECT sal FROM emp WHERE job='MANAGER';

SELECT * FROM emp WHERE sal > ANY(SELECT sal FROM emp WHERE
job='MANAGER');
-- solution using multi row sub-query

-- Find all employees whose sal is more than all the managers.
SELECT MAX(sal) FROM emp WHERE job='MANAGER';

SELECT * FROM emp WHERE sal > (SELECT MAX(sal) FROM emp WHERE
job='MANAGER');
-- solution using single row sub-query

SELECT sal FROM emp WHERE job='MANAGER';
```

```SQL
SELECT * FROM emp WHERE sal > ALL(SELECT sal FROM emp WHERE
job='MANAGER');
-- solution using multi row sub-query
```

```SQL
-- Find all depts which contain at least one employee.
SELECT deptno FROM emp;

SELECT * FROM dept WHERE deptno = ANY(SELECT deptno FROM emp);

SELECT * FROM dept WHERE deptno IN (SELECT deptno FROM emp);

-- Find all depts which doesn't contain any employee.
SELECT * FROM dept WHERE deptno != ALL(SELECT deptno FROM emp);

SELECT * FROM dept WHERE deptno NOT IN (SELECT deptno FROM emp);

SELECT * FROM dept WHERE deptno != ANY(SELECT deptno FROM emp);
-- wrong result
```

Co-related sub-query

```SQL
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;

CREATE TABLE dept(deptno INT(4), dname VARCHAR(40), loc VARCHAR(40));

INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
INSERT INTO dept VALUES (30,'SALES','CHICAGO');
INSERT INTO dept VALUES (40,'OPERATIONS','BOSTON');

CREATE TABLE emp(empno INT(4), ename VARCHAR(40), job VARCHAR(40), mgr
INT(4), hire DATE, sal DECIMAL(8,2), comm DECIMAL(8,2), deptno
INT(4));
```

```
INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-
17',800.00,NULL,20);
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-
20',1600.00,300.00,30);
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-
22',1250.00,500.00,30);
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-
02',2975.00,NULL,20);
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-
28',1250.00,1400.00,30);
INSERT INTO emp VALUES (7698,'BLAKE','MANAGER',7839,'1981-05-
01',2850.00,NULL,30);
INSERT INTO emp VALUES (7782,'CLARK','MANAGER',7839,'1981-06-
09',2450.00,NULL,10);
INSERT INTO emp VALUES (7788,'SCOTT','ANALYST',7566,'1982-12-
09',3000.00,NULL,20);
INSERT INTO emp VALUES (7839,'KING','PRESIDENT',NULL,'1981-11-
17',5000.00,NULL,10);
INSERT INTO emp VALUES (7844,'TURNER','SALESMAN',7698,'1981-09-
08',1500.00,0.00,30);
INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,'1983-01-
12',1100.00,NULL,20);
INSERT INTO emp VALUES (7900,'JAMES','CLERK',7698,'1981-12-
03',950.00,NULL,30);
INSERT INTO emp VALUES (7902,'FORD','ANALYST',7566,'1981-12-
03',3000.00,NULL,20);
INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,'1982-01-
23',1300.00,NULL,10);

SELECT * FROM dept WHERE deptno IN (SELECT DISTINCT deptno FROM emp);

SELECT * FROM dept d WHERE d.deptno IN (SELECT e.deptno FROM emp e
WHERE e.deptno = d.deptno);

SELECT * FROM dept d WHERE EXISTS (SELECT e.deptno FROM emp e WHERE
e.deptno = d.deptno);

SELECT @@optimizer_switch;
SET @@optimizer_switch='materialization=off';
```

```
SET @@optimizer_switch='subquery_materialization_cost_based=off';
SET @@optimizer_switch='block_nested_loop=off';
SET @@optimizer_switch='semijoin=off';

-- costs may differ with platform (OS) and mysql version.

EXPLAIN FORMAT=JSON
SELECT * FROM dept WHERE deptno IN (SELECT DISTINCT deptno FROM emp);

EXPLAIN FORMAT=JSON
SELECT * FROM dept d WHERE d.deptno IN (SELECT e.deptno FROM emp e
WHERE e.deptno = d.deptno);

EXPLAIN FORMAT=JSON
SELECT * FROM dept d WHERE EXISTS (SELECT e.deptno FROM emp e WHERE
e.deptno = d.deptno);
```

```SQL
SELECT * FROM dept d WHERE NOT EXISTS (SELECT e.deptno FROM emp e
WHERE e.deptno = d.deptno);
```

Sub-Query

* Query within query.
    * SELECT within SELECT.
    * In MySQL, sub-query is allowed in UPDATE & DELETE also.
        * SELECT in UPDATE & SELECT in DELETE.
        * Cannot UPDATE/DELETE the table on which sub-query is doing SELECT.

```SQL
-- delete employee with highest salary.
DELETE FROM emp WHERE sal = (SELECT MAX(sal) FROM emp);
-- error: not allowed in MySQL

-- update employee with highest salary to decrease his sal by 500.
UPDATE emp SET sal = sal - 500  WHERE sal = (SELECT MAX(sal) FROM
emp);
-- error: not allowed in MySQL
```

```SQL
-- update dname='TRAINING' in which there are no employees.
UPDATE dept SET dname='TRAINING' WHERE deptno NOT IN (SELECT deptno
FROM emp);
-- allowed:

SELECT * FROM dept;

-- delete depts in which there are no employees.
DELETE FROM dept WHERE deptno NOT IN (SELECT deptno FROM emp);

SELECT * FROM dept;
```

Derived Tables

```SQL
-- find the max of avg sal per job.
SELECT job, AVG(sal) FROM emp
GROUP BY job;

SELECT AVG(sal) FROM emp
GROUP BY job
ORDER BY 1 DESC
LIMIT 1;

-- using derived table
SELECT job, AVG(sal) avgsal FROM emp
GROUP BY job;

SELECT MAX(avgsal) FROM
(SELECT job, AVG(sal) avgsal FROM emp
GROUP BY job) AS jobsal;
-- jobsal is alias for "SELECT job, AVG(sal) avgsal FROM emp GROUP BY
job" query result => derived table
```

Views

- RDBMS view represents view (projection) of the data.
- View is based on SELECT statement.
- Typically it is restricted view of the data (limited rows or columns) from one or more tables (joins and/or sub-queries) or summary of the data (grouping).
- Data of view is not stored on server hard-disk; but its SELECT statement is stored in compiled form. It speed up execution of view.
- Views are of two types: Simple view and Complex view
- Usually if view contains computed columns, group by, joins or sub-queries, then the views are said to be complex. DML operations are not supported on these views.
- DML operations on view affects underlying table.
- View can be created with CHECK OPTION to ensure that DML operations can be performed only the data visible in that view.
- Views can be differentiated with: SHOW FULL TABLES.
- Views can be dropped with DROP VIEW statement.
- View can be based on another view.
- Applications of views
    - Security: Providing limited access to the data.
    - Hide source code of the table.
    - Simplifies complex queries

```
# Views
* syntax: CREATE VIEW viewname AS SELECT ...;

```SQL
  • CREATE VIEW v1_jobsal AS
    SELECT job, AVG(sal) avgsal FROM emp
    GROUP BY job;

  • SHOW TABLES;
  • DESCRIBE v1_jobsal;
  • SELECT * FROM v1_jobsal;
  • SELECT * FROM v1_jobsal ORDER BY avgsal DESC;
  • SELECT MAX(avgsal) FROM v1_jobsal;
```
```

```SQL
CREATE VIEW v2_emp AS
SELECT * FROM emp;
-- v2_emp = emp

SELECT * FROM v2_emp;

CREATE VIEW v3_emp AS
SELECT empno, ename, sal, comm FROM emp;
-- v3_emp = view of emps with limited columns

SELECT * FROM v3_emp;

CREATE VIEW v4_emp AS
SELECT empno, ename, sal, comm, sal + IFNULL(comm,0.0) income FROM
emp;
-- v4_emp = view of emps with limited columns + computed columns

DESC v4_emp;

SELECT * FROM v4_emp;

CREATE VIEW v5_emp AS
SELECT * FROM emp WHERE sal > 2000;
-- v5_emp = view of emps with limited rows

DESC v5_emp;

SELECT * FROM v5_emp;

CREATE VIEW v6_emp AS
SELECT e.empno, e.ename, e.job, e.deptno, e.sal, e.comm, d.dname,
d.loc FROM emp e
INNER JOIN dept d ON e.deptno = d.deptno;
-- v6_emp = joined view of emp and dept.

SELECT * FROM v6_emp;
```

* If DML operations are performed on main table(s), they will automatically reflect in the view.

```SQL
SELECT * FROM v6_emp;

INSERT INTO dept VALUES (40, 'OPERATIONS', 'BOSTON');

INSERT INTO emp(empno,ename,job,sal,comm,deptno) VALUES(1000, 'STEVE', 'DIRECTOR', 4500.00, NULL, 40);

SELECT * FROM emp;

SELECT * FROM v6_emp;

SELECT * FROM v1_jobsal;
```

* Changes done in view will be reflected in main table (if possible).
    * Simple views --> WHERE, ORDER, LIMIT, ...

```SQL
SELECT * FROM v3_emp;

INSERT INTO v3_emp VALUES (1001, 'BILL', 3500.00, 0.0);

SELECT * FROM v3_emp;

SELECT * FROM emp;
```

* Changes done in view will not be reflected in main table (if not possible).
    * Complex views --> Computed columns, GROUP BY, JOIN, Sub-queries.

```SQL
SELECT * FROM v1_jobsal;

DELETE FROM v1_jobsal WHERE job='MANAGER';
-- error: not allowed
```

```SQL
INSERT INTO v1_jobsal VALUES ('TRAINER', 7000.0);
-- error: not allowed
```

```SQL
SELECT * FROM v5_emp;
-- view of emps where sal > 2000.

INSERT INTO v5_emp(empno,ename,job,sal) VALUES(1003, 'CHEN', 'GUARD',
1200);

SELECT * FROM v5_emp;

SELECT * FROM emp;
```

* "WITH CHECK OPTION" check WHERE clause condition while executing DML operations on views. DML operations are allowed only when condition holds true.

```SQL
CREATE VIEW v7_emp AS
SELECT * FROM emp WHERE sal > 2000
WITH CHECK OPTION;

INSERT INTO v7_emp(empno,ename,job,sal) VALUES(1004, 'ROD', 'GUARD',
1300);
-- error: check option failed

INSERT INTO v7_emp(empno,ename,job,sal) VALUES(1005, 'JOHN', 'GUARD',
2300);
```

```SQL
SHOW TABLES;

SHOW FULL TABLES;

SHOW FULL TABLES WHERE Table_type='VIEW';
```

```

```SQL
DROP VIEW v5_emp;

SHOW FULL TABLES WHERE Table_type='VIEW';
```

* If original table is deleted, accessing view results in error.
* Programmer should delete views before the table.

```SQL
SELECT * FROM salgrade;

CREATE VIEW v_salgrade AS
SELECT * FROM salgrade;

SELECT * FROM v_salgrade;

DROP TABLE salgrade;

SHOW FULL TABLES WHERE Table_type='VIEW';

SELECT * FROM v_salgrade;
-- error: original table is dropped
```

```SQL
SELECT * FROM v6_emp;

CREATE VIEW v8_emp AS
SELECT ename, dname FROM v6_emp;
-- view based on another view

SELECT * FROM v8_emp;

SHOW CREATE VIEW v8_emp;

SHOW CREATE VIEW v6_emp;
```
```

```SQL
USE sales;

SELECT * FROM customers;

SELECT * FROM salespeople;

SELECT * FROM orders;

DROP VIEW IF EXISTS salesview;

CREATE VIEW salesview AS
SELECT c.cnum, c.cname, c.city ccity, c.rating, s.snum, s.sname,
s.comm, s.city scity, o.onum, o.amt, o.odate
FROM orders o
INNER JOIN customers c ON o.cnum = c.cnum
INNER JOIN salespeople s ON o.snum = s.snum;

SELECT * FROM salesview;

-- print customers and salespeople living in same city.
SELECT * FROM salesview WHERE ccity = scity;
```

RDBMS Security

* Login with "root" user.
    * terminal> mysql -u root -p

```SQL
SHOW DATABASES;

USE mysql;

SHOW TABLES;

DESCRIBE user;

SELECT user, host FROM user;

CREATE USER pmgr@'%' IDENTIFIED BY 'pmgr';
```

```
-- pmgr user created with pmgr password and he can login from any
machine in the network.

GRANT ALL PRIVILEGES ON dacdb.* TO pmgr@'%' WITH GRANT OPTION;

FLUSH PRIVILEGES;

SELECT user, host FROM user;

CREATE USER developer1@'%' IDENTIFIED BY 'developer1';

CREATE USER developer2@'%' IDENTIFIED BY 'developer2';

CREATE USER developer3@'%' IDENTIFIED BY 'developer3';

SHOW GRANTS;
-- show permissions for current user.

SHOW GRANTS FOR pmgr@'%';

SHOW GRANTS FOR developer1@'%';

```

* Login with pmgr user from another terminal.
    * terminal> mysql -u pmgr -ppmgr

```SQL
SHOW DATABASES;

USE dacdb;

SHOW TABLES;

GRANT INSERT,UPDATE,DELETE,SELECT ON dacdb.* TO developer1@'%';

GRANT INSERT,UPDATE,DELETE,SELECT ON dacdb.emp TO developer2@'%';

GRANT SELECT ON dacdb.dept TO developer3@'%';
```

```SQL
GRANT SELECT ON dacdb.emp TO developer3@'%';
```

* Login with developer1 user from another terminal.

```SQL
SHOW DATABASES;

USE dacdb;

SHOW TABLES;

GRANT SELECT ON dacdb.books TO developer3@'%';
-- error: developer1 doesn't have GRANT OPTION.
```

* Login with developer3 user from another terminal.

```SQL
SHOW DATABASES;

USE dacdb;

SHOW TABLES;

SHOW GRANTS;

DELETE FROM dept;
-- error
```

* With root login

```SQL
SHOW GRANTS FOR developer2@'%';

REVOKE DELETE ON dacdb.emp FROM developer2@'%';

SHOW GRANTS FOR developer2@'%';
```

Data Control Language

- Permissions are given to user using GRANT command.
    - GRANT CREATE TABLE TO user@host;
    - GRANT CREATE TABLE, CREATE VIEW TO user1@host, user2@host;
    - GRANT SELECT ON db.table TO user@host;
    - GRANT SELECT, INSERT, UPDATE ON db.table TO user@host;
    - GRANT ALL ON db.* TO user@host;
- By default one user cannot give permissions to other user. This can be enabled using WITH GRANT OPTION.
    - GRANT ALL ON *.* TO user@host WITH GRANT OPTION;
- Permissions for the user can be listed using SHOW GRANTS command.
- Permissions assigned to any user can be withdrawn using REVOKE command.
    - REVOKE SELECT, INSERT ON db.table FROM user@host;
- Security is built-in feature of any RDBMS. It is implemented in terms of permissions (a.k.a. privileges).
- There are two types of privileges.
- System privileges
- Privileges for certain commands i.e. CREATE TABLE, CREATE USER, CREATE TRIGGER, ...
- Typically these privileges are given to the database administrator. (MySQL root login).
- Object privileges
- RDBMS RbjecWV aUe WabOe, YieZ, VWRUed SURcedXUe, fXQcWiRQ, WUiggeUV, «
- Can perform operations on the objects i.e. INSERT, UPDATE, DELETE, SELECT, CALL, ...
- Typically these privileges are given to the database users

MySQL Programming

- RDBMS Programming is an ISO standard ± part of SQL standard ± since 1992.
- SQL/PSM stands for Persistent Stored Module.
- Inspired from PL/SQL - Programming language of Oracle.
- PSM allows writing programs for RDBMS. The program contains set of SQL statements along with programming constructs e.g. variables, if-else, loops, case, ...
- PSM is a block language. Blocks can be nested into another block.
- MySQL program can be a stored procedure, function or trigger.
- MySQL PSM program is written by db user (programmers).
- It is submitted from client, server check syntax & store them into db in compiled form.
- The program can be executed by db user when needed.
- Since programs are stored on server in compiled form, their execution is very fast.
- All these programs will run in server memory.

Stored Procedure

- Stored Procedure is a routine. It contains multiple SQL statements along with programming constructs.
- Procedure doesn't return any value (like void fns in C).
- Procedures can take zero or more parameters.
- Procedures are created using CREATE PROCEDURE and deleted using DROP PROCEDURE.
- Procedures are invoked/called using CALL statement.
- Result of stored procedure can be
    - returned via OUT parameter.
    - inserted into another table.
    - produced using SELECT statement (at end of SP).
- Delimiter should be set before writing SQL query

```
CREATE TABLE result(v1 DOUBLE, v2 VARCHAR(50));

DELIMITER $$

CREATE PROCEDURE sp_hello()
BEGIN
    INSERT INTO result VALUES(1, 'Hello World');
END;
$$

DELIMITER ;

CALL sp_hello();

SELECT * FROM result;
```

```
① -- 01_hello.sql (using editor)
  DROP PROCEDURE IF EXISTS sp_hello;
  DELIMITER $$
  CREATE PROCEDURE sp_hello()
  BEGIN
      SELECT 1 AS v1, 'Hello World' AS v2;
  END;
  $$
  DELIMITER ;

②
  SOURCE /path/to/01_hello.sql

③
  CALL sp_hello();
```

VARIABLES

```
DECLARE varname DATATYPE;
DECLARE varname DATATYPE DEFAULT init_value;
SET varname = new_value;
SELECT new_value INTO varname;
SELECT expr_or_col INTO varname FROM table_name;
```

PARAMETERS

```
CREATE PROCEDURE sp_name(PARAMTYPE p1 DATATYPE)
BEGIN
    …
END;

-- IN param: Initialized by calling program.
-- OUT param: Initialized by called procedure.
-- INOUT param: Initialized by calling program and
modified by called procedure
-- OUT & INOUT param declared as session variables.

CREATE PROCEDURE sp_name(OUT p1 INT)
BEGIN
    SELECT 1 INTO p1;
END;

SET @res = 0;
CALL sp_name(@res);
SELECT @res;
```

IF-ELSE

```
IF condition THEN
    body;
END IF;
----------------------
IF condition THEN
    if-body;
ELSE
    else-body;
END IF;
----------------------
IF condition THEN
    if1-body;
ELSE
    IF condition THEN
        if2-body;
    ELSE
        else2-body;
    END IF;
END IF;
----------------------
IF condition THEN
    if1-body;
ELSEIF condition THEN
    if2-body;
ELSE
    else-body;
END IF;
```

LOOPS

```
WHILE condition DO
    body;
END WHILE;
------------------
REPEAT
    body;
UNTIL condition
END REPEAT;
------------------
label: LOOP
IF condition THEN
        ...
        LEAVE label;
    END IF;
    ...
END LOOP;
```

CASE-WHEN

```
CASE
WHEN condition THEN
      body;
WHEN condition THEN
      body;
ELSE
      body;
END CASE;
```

SHOW PROCEDURE

```
SHOW PROCEDURE STATUS
LIKE 'sp_name';

SHOW CREATE PROCEDURE sp_name;
```

DROP PROCEDURE

```
DROP PROCEDURE
IF EXISTS sp_name;
```

• Stored Functions are MySQL programs like stored procedures.
• Functions can be having one or more parameters. MySQL allows only IN params.
• Functions must return some value using RETURN statement.
• Function entire code is stored in system table.
• Like procedures, functions allows statements like local variable declarations, if-else, case, loops, etc. One function can invoke another function/procedure and vice-versa. The functions can also be recursive.
• There are two types of functions: DETERMINISTIC and NOT DETERMINISTIC.

CREATE FUNCTION

```
CREATE FUNCTION fn_name(p1 TYPE)
RETURNS TYPE
[NOT] DETERMINISTIC
BEGIN
   body;
   RETURN value;
END;
```

SHOW FUNCTION

```
SHOW FUNCTION STATUS LIKE 'fn_name';

SHOW CREATE FUNCTION fn_name;
```

DROP FUNCTION

```
DROP FUNCTION IF EXISTS fn_name;
```

• Exceptions are runtime problems, which may arise during execution of stored procedure, function or trigger.
• Required actions should be taken against these errors.
• SP execution may be continued or stopped after handling exception.
• MySQL error handlers are declared as:
• DECLARE action HANDLER FOR condition handler_impl;
• The action can be: CONTINUE or EXIT.
• The condition can be:
• MySQL error code: e.g. 1062 for duplicate entry.
• SQLSTATE value: e.g. 23000 for duplicate entry, NOTFOUND for end-of-cursor.
• Named condition: e.g. DECLARE duplicate_entry CONDITION FOR 1062;
• The handler_impl FaQ bH: SLQJOH OLQHU RU PSM bORFN L.H. BEGIN « END;

MySQL Triggers

- Triggers are supported by all standard RDBMS like Oracle, MySQL, etc.
- Triggers are not supported by WEAK RDBMS like MS-Access. SQLite.
- Triggers are not called by client's directly, so they don't have args & return value.
- Trigger execution is caused by DML operations on database.
  - BEFORE/AFTER INSERT, BEFORE/AFTER UPDATE, BEFORE/AFTER DELETE.
- Like SP/FN, Triggers may contain SQL statements with programming constructs. They may also call other SP or FN.
- However COMMIT/ROLLBACK is not allowed in triggers. They are executed in same transaction in which DML query is executed.

CREATE TRIGGER

```
CREATE TRIGGER trig_name
AFTER|BEFORE dml_op ON table
FOR EACH ROW
BEGIN
   body;
   -- use OLD & NEW keywords
   -- to access old/new rows.
   -- INSERT triggers - NEW rows.
   -- DELETE triggers - OLD rows.
END;
```

UPDATE triggers

SHOW TRIGGERS

```
SHOW TRIGGERS FROM db_name;
```

DROP TRIGGER

```
DROP TRIGGER trig_name;
```

- Applications of triggers:
  - Maintain logs of DML operations (Audit Trails).
  - Data cleansing before insert or update data into table. (Modify NEW value).
  - Copying each record AFTER INSERT into another table to maintain "Shadow table".
  - Copying each record AFTER DELETE into another table to maintain "History table".
  - Auto operations of related tables using cascading triggers.

CHECK constraint for checking values.

SET NEW.ename = UPPER(OLD.ename);

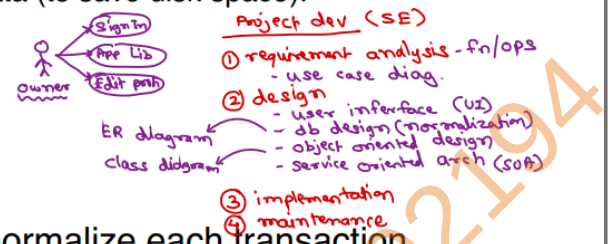
emp → sal_history

- Cascading triggers
  - One trigger causes execution of 2nd trigger, 2nd trigger causes execution of 3rd trigger and so on.
  - In MySQL, there is no upper limit on number of levels of cascading.
  - This is helpful in complicated business processes.

- Mutating table error
  - If cascading trigger causes one of the earlier trigger to re-execute, "mutating table" error is raised.
  - This prevents infinite loop and also rollback the current transaction.

Normalization

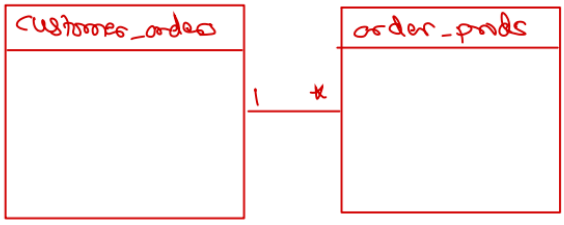
- Concept of table design: Table, Structure, Data Types, Width, Constraints, Relations.
- Goals:
  - Efficient table structure.
  - Avoid data redundancy i.e. unnecessary duplication of data (to save disk space).
  - Reduce problems of insert, update & delete.
- Done from input perspective.
- Based on user requirements.
- Part of software design phase.
- View entire appln on per transaction basis & then normalize each transaction separately.
- Transaction Examples:
  - Banking, Rail Reservation, Online Shopping.



- For given transaction make list of all the fields.
- Strive for atomicity.
- Get general description of all field properties.
- For all practical purposes we can have a single table with all the columns. Give meaningful names to the table.
- Assign datatypes and widths to all columns on the basis of general desc of fields properties.
- Remove computed columns.
- Assign primary key to the table.
- At this stage data is in un-normalized form.
- UNF is starting point of normalization.

- UNF suffers from
  - Insert anomaly :- Some data need to be inserted repeatedly.
  - Update anomaly :- Some changes might be done at multiple records.
  - Delete anomaly :- Some of undesired data may be deleted.

- 1. Remove repeating group into a new table.
- 2. Key elements will be PK of new table.
- 3. (Optional) Add PK of original table to new table to give us Composite PK.
  - Repeat steps 1-3 infinitely -- to remove all repeating groups into new tables.
  - This is **1-NF**. No repeating groups present here. One to Many relationship between two tables.

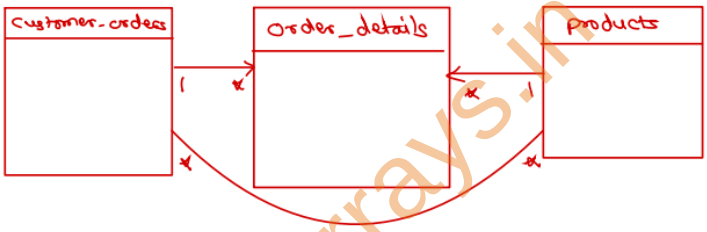


- 4. Only table with composite PK to be examined.
- 5. Those columns that are not dependent on the entire composite PK, they are to be removed into a new table.
- 6. The key elements on which the non-key elements were originally dependent, it is to be added to the new table, and it will be the PK of new table.
  - Repeat steps 4-6 infinitely -- to separate all non-key elements from all tables with composite primary key.
  - This is **2-NF**. Many-to-Many relationship.

Many to many

One order has many products.

One product is purchased in many orders.

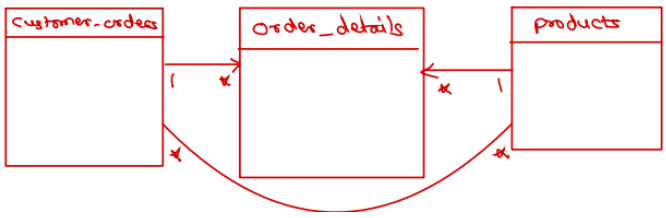


- 4. Only table with composite PK to be examined.
- 5. Those columns that are not dependent on the entire composite PK, they are to be removed into a new table.
- 6. The key elements on which the non-key elements were originally dependent, it is to be added to the new table, and it will be the PK of new table.
  - Repeat steps 4-6 infinitely -- to separate all non-key elements from all tables with composite primary key.
  - This is **2-NF**. Many-to-Many relationship.

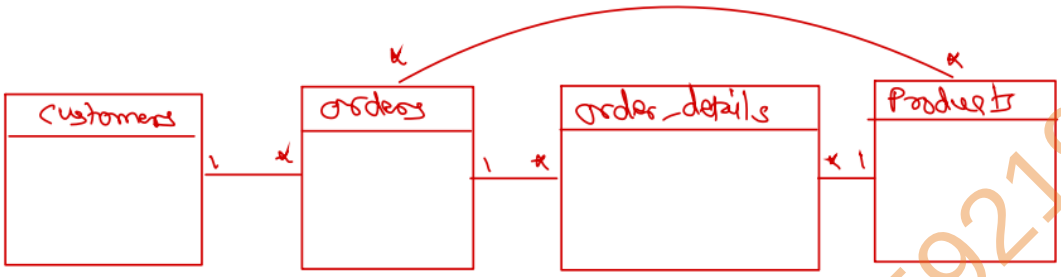
Many to many

One order has many products.

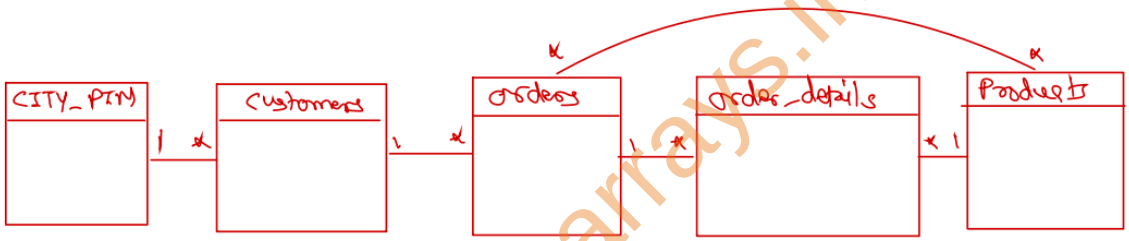
One product is purchased in many orders.



- 7. Only non-key elements are examined for inter-dependencies.
- 8. Inter-dependent cols that are not directly related to PK, they are to be removed into a new table.
- 9. (a) Key ele will be PK of new table.
- 9. (b) The PK of new table is to be retained in original table for relationship purposes.
  - Repeat steps 7-9 infinitely to examine all non-key eles from all tables and separate them into new table if not dependent on PK.
  - This is **3-NF**.



  - To ensure data consistency (no wrong data entered by end user).
  - Separate table to be created of well-known data. So that min data will be entered by the end user.
  - This is BCNF or 4-NF.



De-normalization

- Normalization will yield a structure that is non-redundant.
- Having too many inter-related tables will lead to complex and inefficient queries.
- To ensure better performance of analytical queries, few rules of normalization can be compromised.
- This process is de-normalization.

Codd's rule

• Proposed by Edgar F. Codd ± pioneer of the RDBMS ± in 1980.
• If any DBMS follow these rules, it can be considered as RDBMS.
• The 0th rule is the main rule known as "The foundation Rule".
• For any system that is advertised as, or claimed to be, a relational data base management system, that system must be able to manage data bases entirely through its relational capabilities.
• The rest of rules can be considered as elaboration of this foundation rule.

Rule 1: The information rule:

All information in a relational data base is represented explicitly at the logical level and in exactly one way ± by values in tables.

Rule 2: The guaranteed access rule:

Each and every datum (atomic value) in a relational data base is guaranteed to be logically accessible by resorting to a combination of table name, primary key value and column name.

Rule 3: Systematic treatment of null values:

Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type.

Rule 4: Dynamic online catalog based on the relational model:

The data base description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.

Rule 5: The comprehensive data sublanguage rule:

A relational system may support several languages. However, there must be at least one language that supports all functionalities of a RDBMS i.e. data definition, data manipulation, integrity constraints, transaction management, authorization.

Rule 6: The view updating rule:

All views that are theoretically updatable are also updatable by the system.

Rule 7: Possible for high-level insert, update, and delete:

The capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update and deletion of data.

Rule 8: Physical data independence:

Application programs and terminal activities remain logically unbroken whenever any changes are made in either storage representations or access methods.

Rule 9: Logical data independence:

Application programs & terminal activities remain logically unbroken when information-preserving changes of any kind that theoretically permit un-impairment are made to the base tables.

Rule 10: Integrity independence:

Integrity constraints specific to a particular relational database must be definable in the relational data sublanguage and storable in the catalog, not in the application programs.

Rule 11: Distribution independence:

The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only.

Rule 12: The non-subversion rule:

If a relational system has a low-level (single-record-at-a-time) language, that low level cannot be used to subvert or bypass the integrity rules and constraints expressed in the higher level relational language (multiple-records-at-a-time).

https://www.youtube.com/@codewitharrays

https://www.instagram.com/codewitharrays/

https://t.me/codewitharrays Group Link: https://t.me/cceesept2023

+91 8007592194 +91 9284926333

codewitharrays@gmail.com

https://codewitharrays.in/project