DBT NOTES (CDAC)

By

Prasad. P. Ingavale

Contents

DAY 1 8
DBT(Database Technology) 8
DBMS VS RDBMS 10
Various RDBMS available: 12
Products of MySQL 13
DAY 2 15
4 Sub Divisions of SQL 15
Rules for Table name:- 16
MySQL Data types: 16
STRING DATA TYPES 17
Char (fixed length) 17
Varchar (variable length) 17
Longtext 17
Binary 18
Varbinary 18
Blob →(Binary Large Object) 18
NUMERIC DATA TYPES 19
Integer types (Exact value) 19
Floating - Point type (Approximate value) 19
Fixed - Point type (Exact value) 19
BOOLEAN DATA TYPE 19
DATE AND TIME DATA TYPES 20
Date 20
Time 20
Datetime 20
Year 20
COMMANDS :- 21
CREATE TABLE 21
INSERT 21
COMMENT 22
SELECT 23
WHERE clause: 25

Relational Operators: 26
Logical Operators: 27
Arithmetic Operators: 28
Alias 29
DISTINCT 31
Installation of MySQL 32
How to create new user? 33
Shortcut for MySQL Command Line Client 33
DAY 3 34
ORDER BY clause 36
Blank-Padded comparison semantics: 39
MySQL–SQL-Special Operators (Like, Between, Any, In): 44
UPDATE: 47
DELETE: 48
DROP 48
TRANSACTION PROCESSING 49
COMMIT 49
ROLLBACK 49
SAVEPOINT: - 51
DAY 4 52
MySQL – SQL – Read and Write consistency 52
What happens when user issues INSERT command ? 53
What happens when user issues DELETE command ? 55
What happens when user issues UPDATE command ? 56
Day 5 62
Character Functions 62
CONCAT() → concatenate (to join) 62
UPPER():converts a string/text to upper-case. 62
LOWER():converts a string/text to lower-case 62
INITCAP():Initial capital, it will convert first letter into upper case and rest of letters into lower case 63
LPAD: Right justification puts blank spaces at the left-hand side 63
RPAD: Left justification puts blank spaces at the right-hand side 63

LTRIM: Removes blank spaces from Left-hand side....64
RTRIM: Removes blank spaces from Right-hand side....64
TRIM: Remove blank spaces from both the sides in MySQL....64
SUBSTRING: Extracts a part of the string....64
REPLACE: Replaces one string with second string....65
INSTR: It stands for Instring, it returns starting position of string....65
Length: returns the length of string....66
ASCII: returns the ascii value of 1st letter....66
CHAR: It returns the character corresponding to ascii value....67
SOUNDEX:....67
Number Functions....67
ROUND(): is used to round a number to a specified number of decimal places. If no specified number of decimal places is provided for round-off, it rounds off the number to the nearest integer.....67
TRUNCATE():- (removes the decimal point numbers), truncates a number to the specified number of decimal places.....68
CEIL(): adds 1 to last number by removing decimal point....68
FLOOR(): removes the decimal and it returns the lower number....68
SIGN(): returns sign of number....69
MOD(): it will return remainder....69
SQRT(): it returns square root for positive numbers only....69
POWER(): is used to find the value of a number raised to the power of another number.....69
ABS(): it always returns a positive numbers....70
Date and Time Functions and Formats (MySQL)....70
sysdate:....70
now....70
adddate :....71
datediff....71
date_add :....71
last_day:....71
day_name:....71
addtime:....71

DAY 6 72
LIST Function 72
IS NULL → (Special Operator) 72
IFNULL 73
GREATEST Function: 73
LEAST Function: 74
CASE expression 75
Environment Functions 76
Group/Aggregate Functions 76
Single Row Function vs Multiple row function 77
SUM(): Returns the sum of the values for the specified column 77
AVG():Returns the average of the values in the specified column. 78
MIN(): Returns the smallest value from the specified column 78
MAX(): Returns the largest value from the specified column 78
COUNT(): Counts the number of rows in each group. 79
COUNT(*): This function counts the number of rows in a given column or expression including NULL and Duplicate values. 79
COUNT-QUERY (counting the number of query hits):- 80
SUMMARY REPORTS: - 82
Restrictions in group functions 82
Group By clause: 83
Difference between Group By and Order By 88
HAVING clause: 89
MATRIX REPORT: - 90
Difference between WHERE clause and HAVING clause 91
In MySQL: - 92
In Oracle: - 92
SQL Order of Execution 93
Order of Execution of SQL Queries Example 94

DAY 7 103
MYSQL - SQL – JOINS (V. imp) 103
5 Types of Joins:- 107
EQUIJOIN (NATURAL JOIN) 107
INEQUIJOIN(NON – EQUI JOIN) 108
OUTER JOIN 109
CARTESIAN JOIN (CROSS JOIN): Returns all records from both tables. 112
SELF JOIN 112
Joining 3 or more tables 115
Types of relationship 116
DAY 8 117
MySQL – SQL – SUB- QUERIES (V. imp) 117
Single Row Subquery 117
Multi-Row Sub-query 123
Using Sub-query in the HAVING clause: - 126
Correlated Sub-query (using the EXISTS operator) 128
MySQL-SQL-Set Operators 130
Pseudo Columns 132
ALTER table (DDL command) 133
DAY 9 136
Difference between DELETE AND TRUNCATE 137
MySQL – SQL – INDEXES (V. imp) 142
Conditions when an index should be created 146
CREATING INDEX 147
COMPOSITE INDEX 148
UNIQUE INDEX 149
Types of Indexes:- 149
MySQL – SQL – CONSTRAINTS (V. imp) 149
PRIMARY KEY constraint 149
COMPOSITE PRIMARY KEY constraint 150
NOT NULL constraint 155
UNIQUE constraint 155
Primary Key VS Unique Key 157

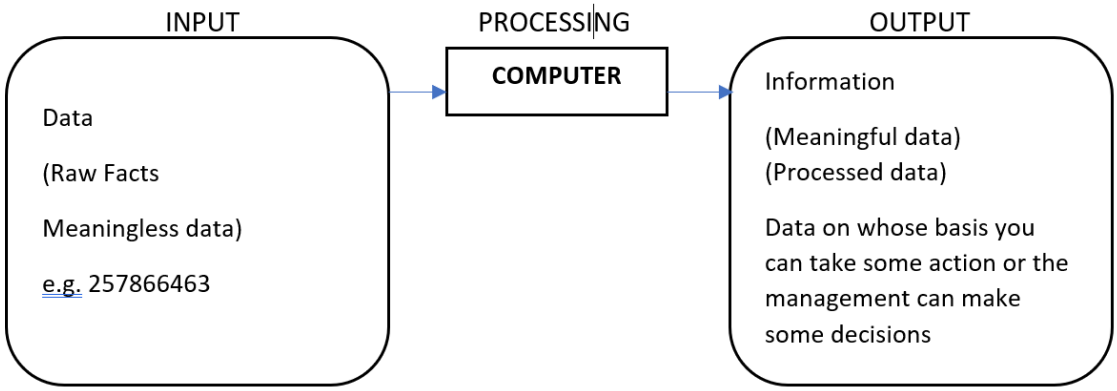
DAY 10 158
FOREIGN KEY constraint: - 158
CHECK constraint 162
DEFAULT 162
MySQL – SQL – PRIVILEGES 163
GRANT 163
REVOKE 164
System tables 164
Architecture for MySQL 165
MySQL – STORED OBJECTS 165
VIEWS 165
WITH CHECK OPTION : 168
MySQL – PL (Programming Language) 170
Temporary Tables 171
DAY 11 174
MySQL – STORED OBJECTS 174
STORED PROCEDURES 174
Need/ Significance of Delimiter 175
COMMENTS 180
Decision making using IF statement 184
CASE statement 187
MySQL – PL – LOOPS 187
While Loop 187
Repeat loop 192
Loop, Leave and Iterate statements 193
Global Variables: - 193
MySQL – PL – Sub-blocks(block within block) 194
DAY 12 197
MySQL - PL – Cursors (V. imp)(Most Imp) 197
Parameters 206
MySQL - STORED OBJECTS 209
STORED FUNCTIONS 209
Interview Question 211

DAY 13 214
DATABASE TRIGGERS (V. imp) 214
INSERT event 215
DELETE event 220
UPDATE event 223
Cascading triggers 224
Mutating table error 224
DAY 14 228
NORMALISATION (V. Imp) 228
Getting ready for Normalisation 228
Steps for Normalisation 230
FIRST NORMAL FORM (FNF) (Single Normal Form) (1 NF) → 230
SECOND NORMAL FORM (SNF) (Double Normal Form) (2 NF) → 231
THIRD NORMAL FORM (TNF) (Triple Normal Form) (3 NF) → 231
FOURTH NORMAL FORM 232
DE-NORMALISATION 232

DAY 1

DBT(Database Technology)

- Database Concepts
- Client-Server Technology
- MySQL (version 8)-is RDBMS(Relational Database Management System)
- Oracle 11g :–
    a. few commands.
    b. Grid Computing
    c. RDBMS + ODBMS(Object Oriented Database Management System)
    d. ORDMS(Object Relational Database Management System)
    (**RDBMS + ODBMS => ORDBMS**)
- Introduction to NoSQL(Not Only SQL)-new technology
- Introduction to MongoDB(version 3.2)-(Mongoose DB)
  (MongoDb is NoSQL technology)



- **Processing** is work done by computer to convert data into information.
- **Database** is collection of large amounts of data.

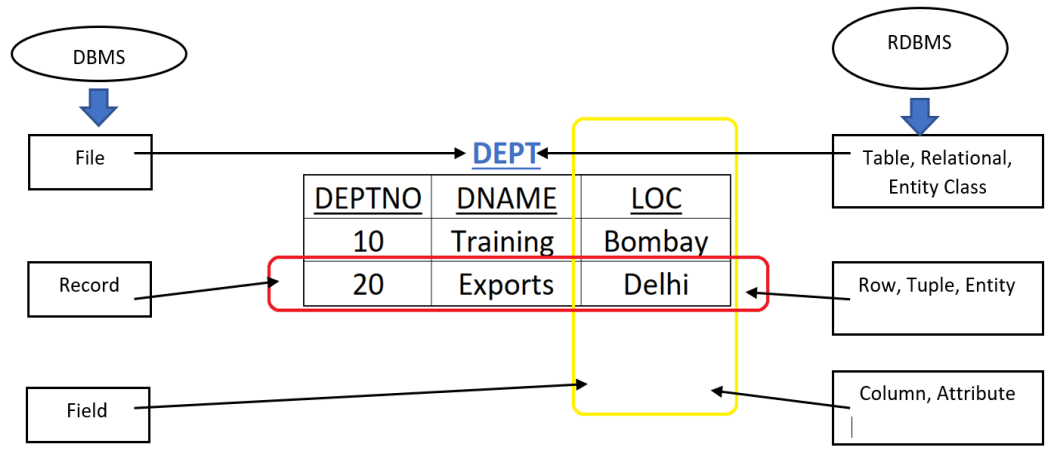
  - **DBMS** is Database management system.
    - It is readymade software that helps you to manage your data. e.g., MS Excel
    - According to ANSI definition collection of programs (readymade software) that allows you to
      - Insert (adding new row, column)
      - Update (modifying row data)
      - Delete
      - Process
    - Various DBMS available: -
      - MS Excel
      - dBase
      - Foxbase
      - Foxpro
      - Clipper
      - Datease
      - Dataflex
      - Advanced
      - Revelation
      - DB Vista
      - Quattro Pro
      - Lotus 1-2-3

NOTE: 53% work in IT industry is done by using MS Excel. MS Excel program is known as Macro(VBA Programming)

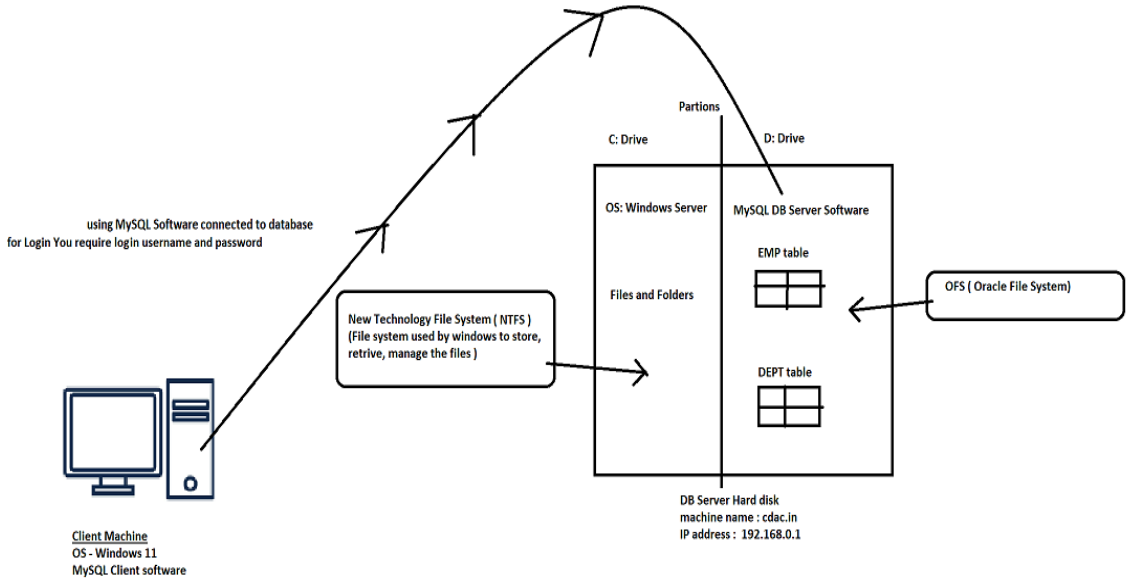
MySQL

MySQL is RDBMS (Relational Database Management System)

DBMS VS RDBMS

	DBMS	RDBMS
	e.g., MS Excel, FoxPro, etc.	e.g., MySQL, Oracle, etc.
1	**Naming Conventions are Different i.e., Nomenclature is Different**	
a	Field	Column, Attribute
b	Record	Rows, Tuple, Entity
c	File	Table, Relational, Entity Class
		
2	Relationship between two files is maintained programmatically	Relationship between two tables can be specified at the time of table creation (e.g., foreign key constraint)
3	More Programming	Less Programming
4	More time is required for software development	Less time is required for software development
5	High network traffic	Low network traffic
6	Slower	Faster
7	More expensive	Cheaper (in terms of hardware cost, network cost, infrastructure cost)
8	Processing on Client machine	Processing on Server machine (Known as Client-Server architecture)

	DBMS	RDBMS
9	Client-Server architecture is not supported	Most of the RDBMS supports Client-Server architecture
10	File level Locking	Row level locking (Table is not a file, internally every row is a file)
11	Not suitable for multi-user	Suitable for multi-user
12	Distributed databases are not supported	Most of the RDBMS support Distributed databases
13	No Security of data • DBMS is dependent on OS for security • DBMS allows access to the data through OS • DBMS allows access to the data outside of the DBMS that created it • Security is not an in-built feature of DBMS	Multiple levels of Security • RDBMS does not allow access to the data through the OS • RDBMS does not allow access to the data outside of the RDBMS that created it A. Logging in security (MySQL username and password) B. Command Level security (Permission to give/issue certain MySQL commands e.g., create table, create function, create procedure, create trigger, create user) C. Object level security (To access the tables and other objects of other users)
14	Various DBMS available: • MS Excel • dBase • FOXBase • FOXPro • Clipper • DataEase • Dataflex • Advanced Revelation • DB Vista • Quattro Pro	Various RDBMS available: • Informix(fastest in terms of processing speed), • Oracle, • Sybase, Oracle, • MS SQL Server, • Ingres, Postgres, • Unify Non-Stop, • DB2, • CICS, • TELON, • IDMS MS Access, • Paradox Vatcom SQL, • MySQL etc.



Various RDBMS available:

Informix:

- Fastest in terms of processing speed.
- Programming has to be done in assembly language

Oracle:

- Most popular RDBMS (because programming is easy)
- Product of Oracle corporation (1977)
- #1 Overall largest software company in the world
- #1 DB largest software company in the world
- 63% of world commercial DB market in the Client Server environment
- 86% of world commercial DB market in the Internet environment
- Works on 113 OS
- 10/10 of top 10 companies in the world use Oracle
- 90% of fortune 500 companies use Oracle

Single User RDBMS: -

- MS Access (good RDBMS from Microsoft)
- Paradox
- Vatcom SQL

Sybase:

- Going down
- Recently acquired by SAP

MS SQL Server

- Good RDBMS from Microsoft (17% of world commercial DB market)
- Only works with WINDOWS OS

MySQL:

- MySQL was launched by a Swedish company in 1995
- Its name is combination of "My", the name of co-founder Michael Widenius daughter and "SQL"
- MySQL was an initially open-source RDBMS
- Part of widely used LAMP open-source web application s/w stack (and other AMP stacks)
- Most of the free software open-source projects that require a RDBMS use MySQL
- E.g., Facebook, Twitter, Google (though not for searches), Flicker, Joomla, Instagram, YouTube, WordPress etc.
- MySQL occupies 42% of world free s/w market
- Sun Microsystems acquired MySQL in 2008
- Oracle Corporation acquired Sun Microsystems in 2010

Various s/w development tools of MySQL

Products of MySQL

1. MySQL database s/w:
   Store table data, retrieve table data, secure table data
2. SQL (Structured Query Language):
   - Commonly pronounced as "Sequel"
   - Create, Drop, Alter
   - Insert, Update, Delete
   - Grant, Revoke, Select
   - Conforms to ANSI standards (e.g., datatypes, 1 char = 1 Byte of storage, etc.)
   - Conforms to ISO standards (for QA)
   - Common for all RDBMS
   - Not a product of MySQL (Common for all RDBMS)
   - Initially founded by IBM (1975-1977)
   - Initially known as RQBE (Relational Query by Example)
   - IBM gave RQBE free of cost to ANSI
   - ANSI renamed RQBE to SQL
   - Now controlled by ANSI (hence SQL is common for all RDBMS)
   - In 2005, ANSI rewrote source of SQL in JAVA (100% JAVA)
3. MySQL-PL
   - MySQL programming language
   - Programming language from MySQL
   - Used for database programming
4. MySQL Command Line Client
   - MySQL Client software
   - Character based (text based)
   - Used to connect to the MySQL database, run SQL commands, run MySQL-PL programs, run MySQL commands

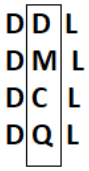
5. MySQL Workbench
    - MySQL Client software
    - GUI based (text based)
    - Used to connect to the MySQL database, run SQL commands, run MySQL-PL programs, run MySQL commands
6. MySQL Connectors
   For database connectivity (JDBC, ODBC, Python, C, C++)
7. MySQL for Excel
   Import, export and edit MySQL data using MS Excel
8. MySQL Enterprise Backup
    - Export and import of table data
    - Used for taking backups of table data (Export) and
    - Used to restore the table data from the backups (Import)
    - Used to transport the tables from one MySQL database to another MySQL database

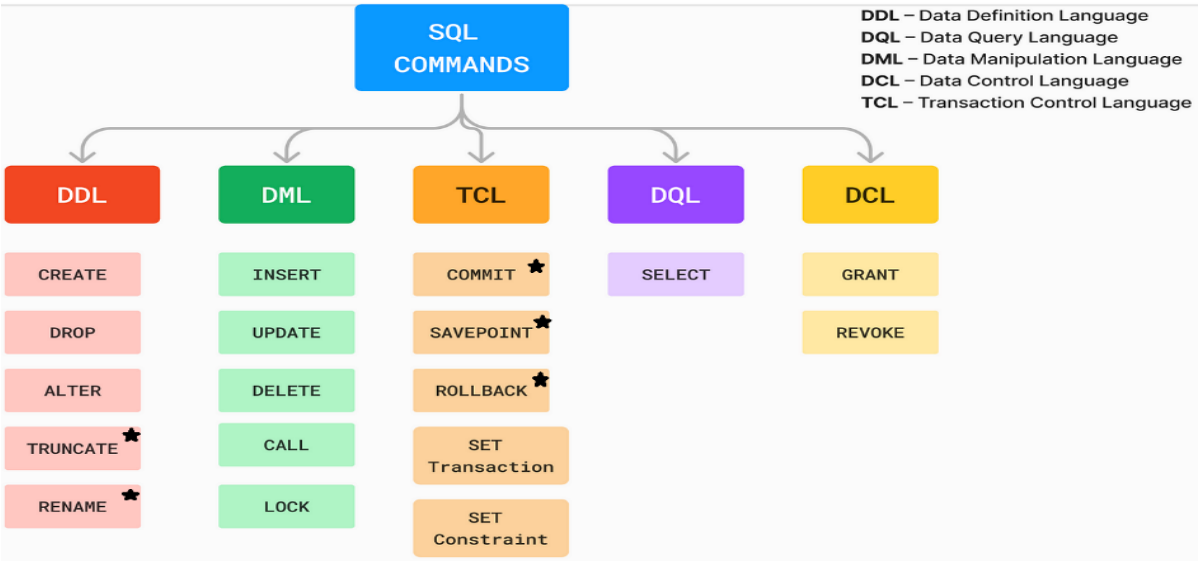
DAY 2

4 Sub Divisions of SQL

[shortcut DMCQ (Delhi Municipal Corporation Q)]

1. **DDL** (Data Definition Language) →(Create, Drop, Alter)
2. **DML** (Data Manipulation Language) →(Insert, Update, Delete)
3. **DCL** (Data Control Language) →(Grant, Revoke)
4. **DQL** (Data Query Language) →(Select)





Extra in MySQL RDBMS and Oracle RDBMS:-

- Not an ANSI standard:-
- 5th component of SQL:-
    - DTL/TCL (Data Transaction Language) / (Transaction Control Language) (Commit, Rollback, Savepoint)

(NOTE: If you **INSERT** a row it is temporary, If you want to make a row permanent you have to say **COMMIT**, If I don't want to save changes I can say **ROLLBACK**)

- DDL (Data Definition Language): - Rename, Truncate

Extra in Oracle RDBMS only:-

DML (Data Manipulation Language) →(Merge, Upsert)

Note:- How many commands are there in SQL ????

Total 16 commands

- 9 commands common for all RDBMS,
- 5 commands extra for MySQL and
- 2 Extra in Oracle only.

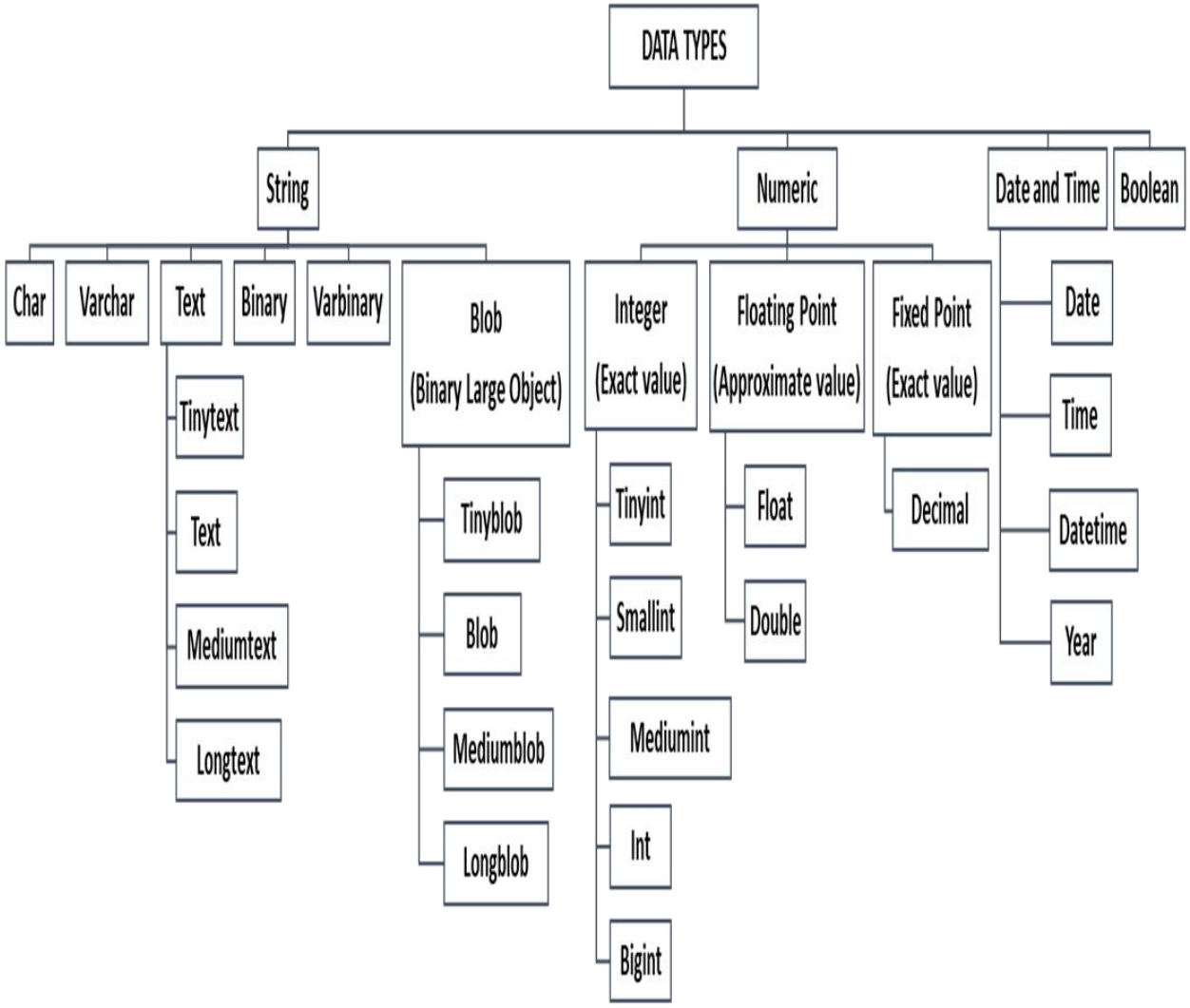
} Oracle Total = 16 commands

Rules for Table name:-

- Max 30 characters allowed.
- A – Z, a – z, 0 – 9 allowed
- Tablename is case insensitive(is not case sensitive)
- Has to begin with an alphabet (e.g EMP)
- Special Characters $, #, _ allowed(e.g EMP_2007)
- In MySQL, to use reserved characters such as # in tablename, enclose it in back-quotes(``) e.g `EMP#`
  (Note: back-quotes (`) symbol is in the upper left corner of your keyboard, just below the escape key and above the tab key.)

- 134 reserved words are not allowed in tablename
- **Above same rules will apply for '*COLUMN NAMES*' and '*VARIABLE NAMES*'**

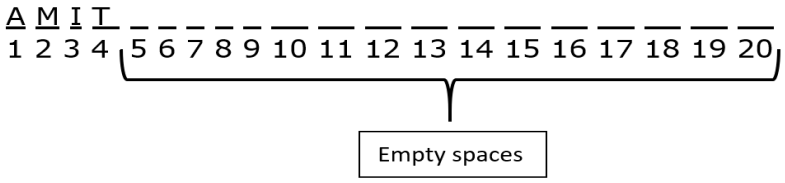
MySQL Data types:



STRING DATA TYPES

Char (fixed length)

- Allows any character
- Could be alphanumeric also
- Maximum 255 character
- Default width is 1
- Fixed length of string
  (for e.g., Ename char(20); and we entered AMIT which is of 4 characters it consumes 4 bytes and 16 spaces are left empty on right side)



- Wastage of HDD
- Searching and retrieval is very fast
- e.g., *ROLL_NO, EMPNO, PANNO, GST_NO, PINCODE, etc.*

Varchar (variable length)

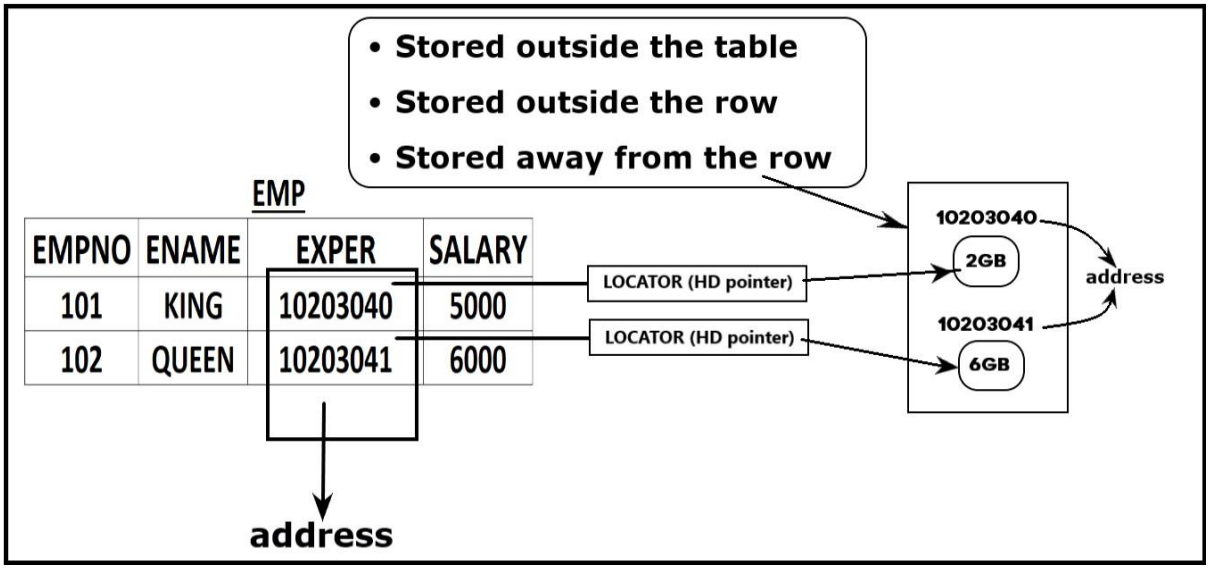
- Allows any character
- Could be alphanumeric also
- Maximum 65,535 characters (64KB – 1),
  (1 character = 1Byte)

(64*1024)-1 = 65536-1=65535

- No Default width (width has to be specified)
- Conserve HDD(varchar is better than in terms of space)
- Searching and retrieval will be slow
- e.g., *ENAME, ADDRESS, CITY etc.*

Longtext

- Allows any character
- Could be alpha numeric also
- Maximum 4,294,967,295 characters (4GB - 1)
- Stored outside the table
- Stored outside the row
- Stored away from the row
- Used for those columns that will be not be used for searching
- Used for those columns that will be used only for storage and display purposes
- e.g., *RESUME, EXPERIENCE, REMARKS, COMMENTS, FEEDBACK, REVIEWS, PRODUCT_DETAILS, etc.*
- MySQL maintains a LOCATOR (HD pointer) from the column to the longtext data
- Width does not have to be specified
- Variable length



Binary

- Fixed length binary string
- Maximum 255 bytes of binary data
- Width need not to be specified
- E.g., BARCODES, FINGERPRINTS, QR_CODES, SIGNATURE

Varbinary

- Variable length binary string
- Maximum 65,535 bytes of binary data
- No default width (width has to be specified)
- Small images, ICONS, etc.

BOTH OF THE ABOVE (BINARY AND VARBINARY) ARE STORED AS CHARACTER STRINGS OF 1's AND 0's

Blob →(Binary Large Object)

a. Tinyblob--------→ (max upto 255 Bytes of binary data)
b. Blob----------→ (max upto 65,535 Bytes of binary data)
c. Mediumblob---→ (max upto 16,777,215 Bytes of binary data)
d. Longblob-------→ (max upto 4,294,967,295 bytes of binary data)
    - Maximum 4,294,967,295 bytes of binary data (4GB -1)
    - Stored outside the table
    - Stored outside the row
    - Stored away from the row
    - MySQL maintains a LOCATOR (HD pointer) from the column to the Longblob data
    - E.g., *PHOTOGRAPHS, IMAGES, GRAPHS, CHARTS, MAPS, SOUND, MUSIC, VIDEOS, etc*
    - This is the multimedia datatypes of MySQL

NUMERIC DATA TYPES

Integer types (Exact value)

- Signed or Unsigned
- By default it is signed

a) Tinyint:- (occupies 1 Byte of storage)
b) Smallint:- (occupies 2 Byte of storage)
c) Mediumint:- (occupies 3 Byte of storage)
d) Int:- (occupies 4 Byte of storage)
e) Bigint:- (occupies 8 Byte of storage)

Floating - Point type (Approximate value)

a) Float:- upto 7 decimals
b) Double:- upto 15 decimals

Fixed - Point type (Exact value)

a) Decimal
    - Stores double as string e.g 653.7
    - Used when it is important to preserve exact precision, for example with monetary data
    - Maximum number of digits is 65

BOOLEAN DATA TYPE

- Logical data type
- True and False evaluates to 1 and 0 respectively.
- E.g., MARITAL_STATUS boolean
- Can insert true, false, 1, or 0
- Output will display 1 or 0

DATE AND TIME DATA TYPES

Date

- **'YYYY-MM-DD'** is the default date format in MySQL
- 1st Jan 1000 AD to 31 st Dec 999 AD
- No problem of Y2K
- **'**YYYY-MM-DD' or 'YY-MM-DD'
  '2023-11-03' or '23-11-03'
  (specifying all 4 digits of year is optional)
- Year values in the range 00-69 are converted to 2000-2069
- Year values in the range 70-99 are converted to 1970-1999
- '23-11-03'
- '1947-08-15'
- '47-08-15'
- 1970 is the known as the Year of the Epoch
- Date1- Date2 → returns the number of days between the 2 dates
  1ST Jan 1000 AD → 1
  2nd Jan 1000 AD → 2
  3rd Jan 1000 AD → 3
  .....
  3rd Nov 2023 AD → 1463218 → number of days since 1ST Jan 1000 AD
- Internally date is stored as a fixed length number and
  Date occupies 7 Bytes of storage

(Note: 1970 is cut off year because UNIX was first operating system in the world was launched in 1970 and in those days the computer calender used to start at 1st Jan 1970 as the mark of respect 1970 is considered as cutt-off year)

Time

- **'hh:mm:ss' or 'HHH:MM:SS'**
- Time values may range from '-838:59:59' to '838:59:59'

Datetime

- **'YYYY-MM-DD hh:mm:ss'**
- '1000-01-01 00:00:00' to '9999-12-31 23:59:59'
- Datetime1 – Datetime2
  (returns the number of days,remainder hours, minutes, and seconds between the two)

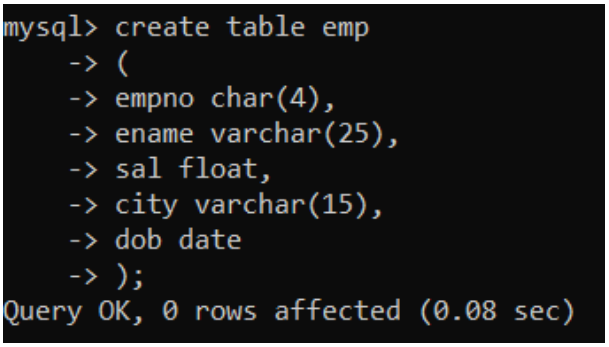
Year

- **YYYY**
- **1901 - 2155**

COMMANDS :-

CREATE TABLE

```
create table emp
(
empno char(4),
ename varchar(25),
sal float,
city varchar(15),
dob date
);
```



Note: Structure of table i.e column sequence never changes, column remains same as in sequence it is entered.

" ; " is known as terminator (denotes the end of command)

"system cls;" to clear terminal

INSERT

For char, varchar and date use '' (open and close single inverted comma)

Easier to write:-

```
insert into emp
values('1', 'Amit', 5000, 'Mumbai', '1995-01-15');
```

Recommended:-

```
insert into emp(empno, sal, ename, city, dob)
values('2', 6000, 'King', 'Delhi', '1990-04-25');
```

Above statement is recommended because of following reasons:

- Flexible
- Readable
- In future if you alter the table and a column, then this **INSERT statement** will continue to work; it will insert a null value for the newly added column

To insert multiple rows using a single INSERT statement

```
insert into emp values
('1', 'Amit', 5000, 'Mumbai', '1995-01-15'),
('2', 'King', 6000, 'Mumbai', '1995-04-25'),
('3', null, 7000, null, null),
('4', 'Atul', null, null, null);
```

To insert data only in specific columns

```
insert into emp(empno, sal)
values('3', 7000);
```

To insert multiple data only in specific columns

```
insert into emp(empno, sal) values
('1', 5000),
('2', 6000),
('3', 7000),
('4', 8000);
```

Inserting Null value

- Null means nothing
- Null value is having ASCII value 0
- Special Treatment given to null value in all RDBMS
- Null value is independent of datatype
- Null value occupies only 1 Byte of storage
- If the row is ending with null values, then all those column will not occupy any space
- Those columnd that are likely to have large number of null values, it is recommended that they should be specified at the end of the table structure, to conserve on the HD space

```
insert into emp
values('4', 'Atul');   <-- ERROR (not enough values)

insert into emp
values('4', 'Atul', null, null, null);

insert into emp
values('5', null, 5000, null, null);
```

COMMENT

MySQL Server supports three comment styles: -

- From a **#** character to the end of the line.
- From a **--** sequence to the end of the line.
  (In MySQL, the -- (double-dash) comment style requires the second dash to be followed by at least one whitespace or control character (such as a space, tab, newline, and so on))
- From a **/* sequence to the following */** sequence, as in the C programming language. This syntax enables a comment to extend over multiple lines because the beginning and closing sequences need not be on the same line.

The following example demonstrates all three comment styles:

```
mysql> SELECT 1+1;     # This comment continues to the end of line
mysql> SELECT 1+1;     -- This comment continues to the end of line
mysql> SELECT 1 /* this is an in-line comment */ + 1;
mysql> SELECT 1+
/*
this is a
multiple-line comment
*/
1;
```

SELECT

It is used to retrieve data from the tables(If you want to view data in table we write SELECT statement)

```
select * from emp;
```

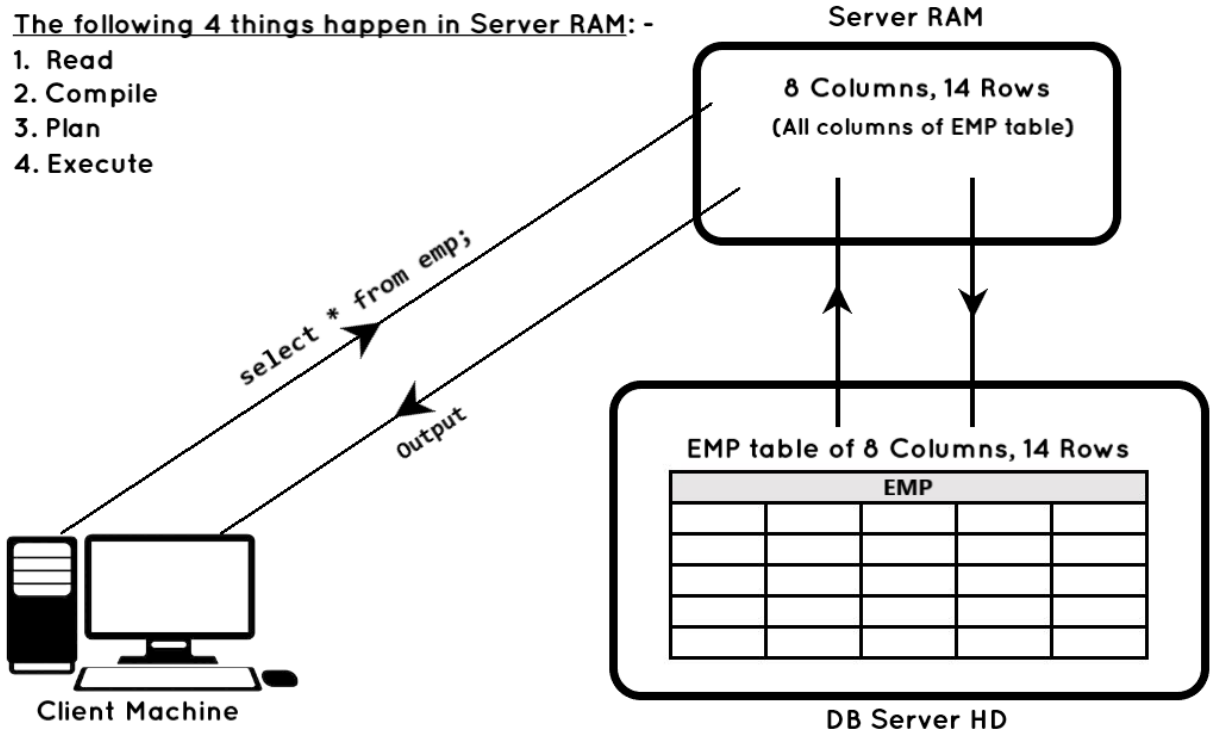
- This statement shows all the row, all the column in the table
- **"*"** is known as metacharacter (all columns)

Once SELECT statement has reached server it first goes to Server RAM and in Server RAM following 4 things happen: -

1. READ → 2. COMPILE →3. PLAN → 4. EXECUTE

1. **Read:** Once SELECT statement has reached server it first is going to read SELECT statement i.e.
   ```
   select *from emp
   ```
2. **Compile:** In this, SELECT statement is converted to machine language i.e., into 0's and 1's
3. **Plan:** It is going to make a plan how to execute a SELECT statement and the plan is as following

Go to Server HD, search for EMP table, once EMP table has been found, all the data(columns) of EMP table will be brought into server RAM and with help of network, output will be sent to client machine and output will be seen on screen



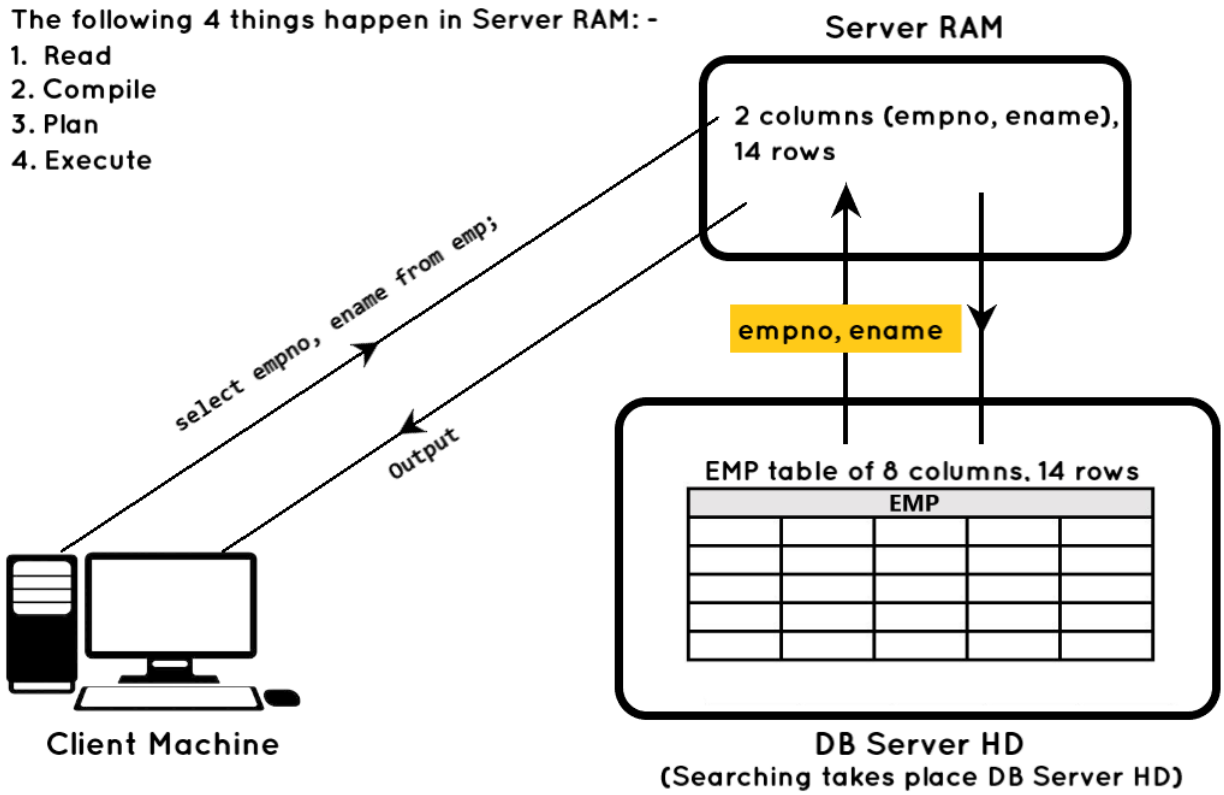
4. **Execute:** It will execute/follow the above plan

- Even if you type SELECT statement 100 times, it will follow Read→ Compile→ Plan→ Execute
- All this process takes place in Server nothing is happening in Client machine, only user has to type SELECT statement and see the output

To Restrict the Columns

```
select empno, ename from emp;
select deptno, job, ename, sal, hiredate from emp;
```

- The positions of columns in SELECT statement, and the position of columns in the table need not be the same
- The positions of columns in SELECT statement, will determine the position of columns in the output (this you will write as per User requirements)



Once SELECT statement has reached server it first goes to Server RAM, in the server RAM it, Read→ Compile→ Plan→ Execute.

Searching takes place in DB server HD and only empno and ename columns with respective to their rows are brought in the Server RAM.

WHERE clause:

- WHERE clause modifies the SELECT statement to look only for row that matches where condition
- The WHERE clause is used to filter records.
- It is used to extract only those records that fulfil a specified condition.

To Restrict the Rows

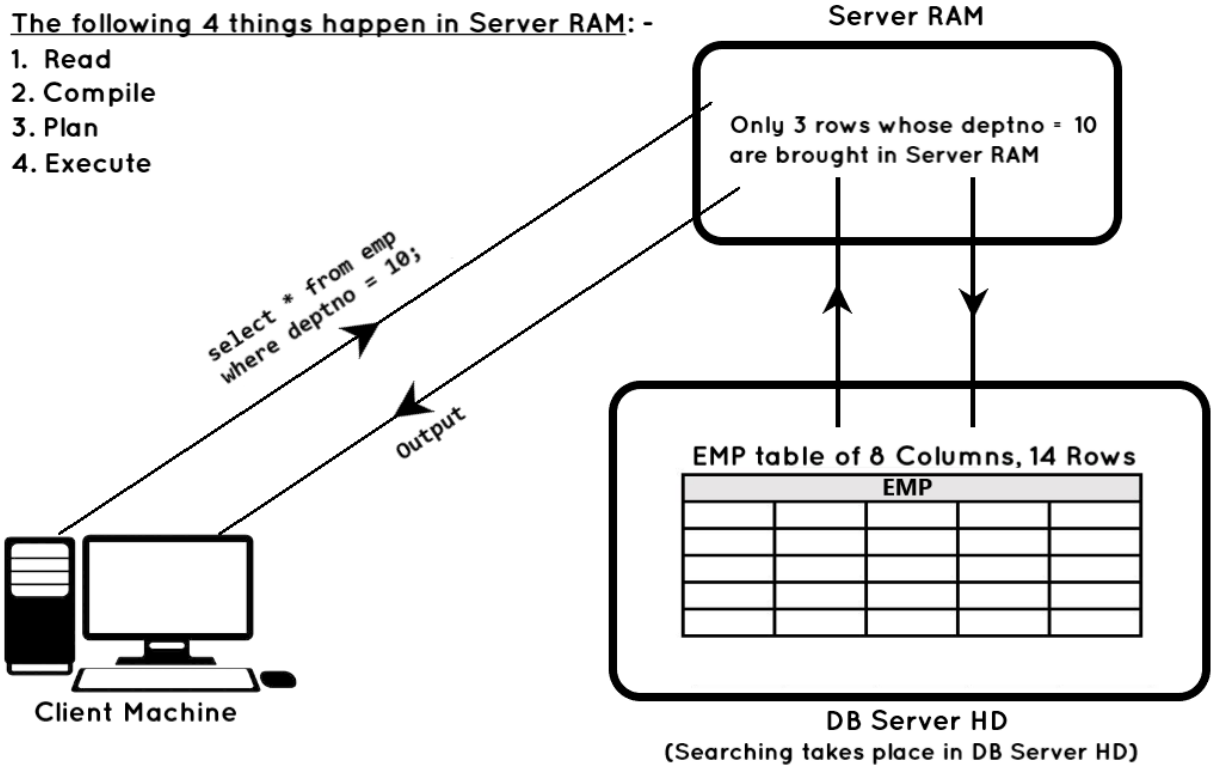
`select * from emp` `where deptno = 10;`	Only those rows will be seen whose deptno is 10 and rest of the rows will be restricted

- WHERE clause is used for restricting rows
- WHERE clause is used for searching.
- Searching takes place in DB server HD
- WHERE clause is used restrict the rows
- WHERE clause is used to retrieve the rows from DB server HD to server RAM

```
select deptno, ename, sal from emp
where deptno = 10;

select * from emp
where deptno = 10;

select * from emp
where sal > 2000;
```



Relational Operators:

Precedence	Relational Operators
1	>
2	>=
3	<
4	<=
5	!= or <> not equal to
6	=

```
select * from emp
where sal > 2000;

select * from emp
where sal > 2000 and sal < 3000;

select * from emp
where sal >= 2000 and sal <= 3000;

select * from emp
where sal > 2000 and sal < 3000;
```

- **WHERE** clause is used to filter the results obtained by the DML statements such as SELECT, UPDATE and DELETE etc.
- We can retrieve the data from a single table or multiple tables (after join operation) using the WHERE clause.

Logical Operators:

Precedence	Logical Operators
1	NOT
2	AND
3	OR

```
select * from emp
where deptno = 10 or sal > 2000 and sal < 3000;
```

1

```
select * from emp
where (deptno = 10 or sal > 2000) and sal < 3000;
```

```
select * from emp
where sal > 2000 or sal < 3000;
```

It will give all rows, instead it is better to use select statement

```
select * from emp
where job = 'MANAGER';
select * from emp
where job = 'manager';        →(works in MySQL and not in ORACLE)
```

- In MySQL and Oracle, when you are inserting, the data is more case – sensitive
- In MySQL, when you are searching, the queries are case – insensitive (more user friendly)
- In Oracle, when you are searching, the queries are case – sensitive (more secure)

```
select * from emp
where job = 'MANAGER' or job = 'CLERK';
```

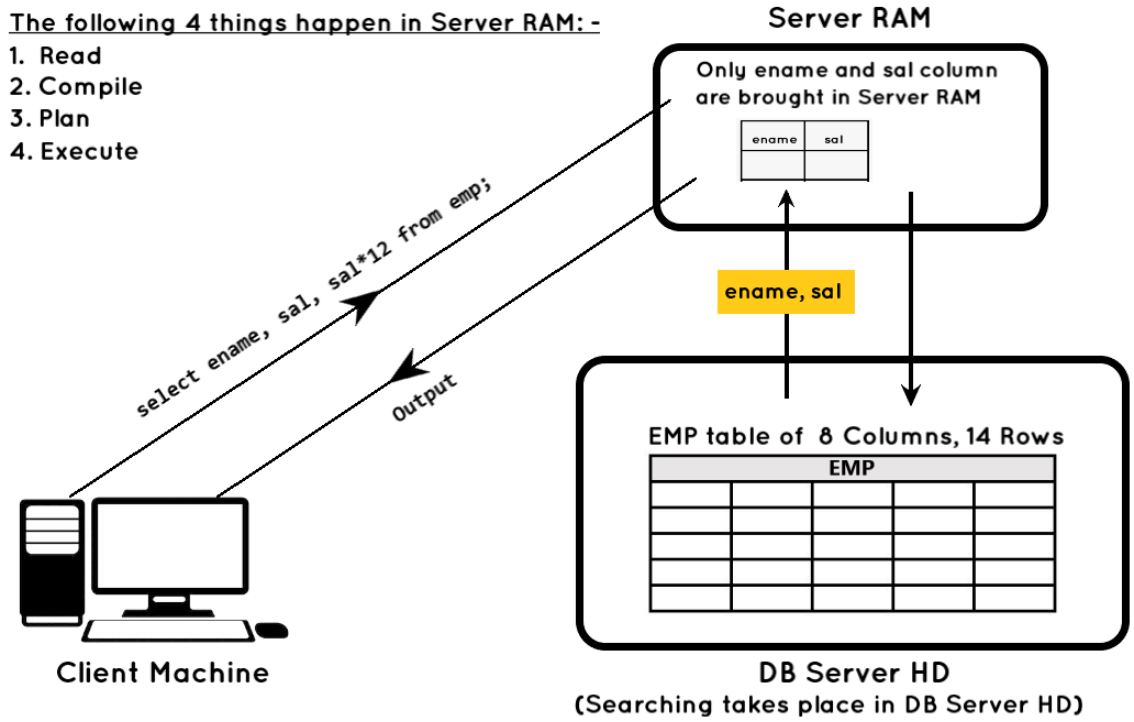
```
select * from emp
where job = 'MANAGER' and job = 'CLERK';
```

Logical Error, no rows will be selected

```
select ename, sal, sal*12 from emp;
```

Here,

- " * " is arithmatic operator (multiplication) and not a metacharacter
- **sal*12** is computed column(derived column), (fake column), (pseudo column)
- it is recommended that computed columns should not be stored in the table, because that would be a wastage of HD space



How internally execution takes place?

STEP 1:

MySQL retrieves the `ename` and `sal` columns of `emp` table from DB Server HD into the Server RAM (**Only *ename* and *sal* column are brought in Server RAM)** then MySQL creates a 2D array in memory to store the `ename` and `sal` columns. MySQL will store ename and sal inside this 2D Array

STEP 2:

Now, MySQL iterates through each row in the 2D array, MySQL uses a FOR loop to go through each row, subsequently calculating `sal*12` by multiplying `sal` by 12 and then adds this value to the result (resultant might be 3D Array) and output will return to client machine

Arithmetic Operators:

Precedence	Operators:
1	() 2+3*4 → 14 (2+3) *4 → 20
2	/
3	*
4	+
5	-

Alias

- SQL aliases are used to give a table, or a column in a table, a temporary name.
- The renaming is just a temporary change and the table name does not change in the original database.
- Aliases are often used to make column names more readable.
- An alias only exists for the duration of that query.
- An alias is created with the AS keyword.
- An alias can be used in a query select list to give a column a different name. You can use the alias in GROUP BY, ORDER BY, or HAVING clauses to refer to the column:

```
SELECT SQRT(a*b) AS root FROM table_name
  GROUP BY root HAVING root > 0;
SELECT id, COUNT(*) AS cnt FROM table_name
  GROUP BY id HAVING cnt > 0;
SELECT id AS 'Customer identity' FROM table_name;
```

```
select ename, sal, sal*12 from emp;
```

```
select ename, sal, sal*12 as "annual"
from emp;
```

sal*12 will be displayed as **annual** in output screen but not in actual table

- Keyword **as** is optional in MySQL and Oracle

```
select ename, sal, sal*12 "annual"
from emp;
```

If you want your alias to contain one or more spaces, like "My Great Products", surround your alias with square brackets or double quotes.
Example:

- Using [square brackets] for aliases with space characters:

```
SELECT ProductName AS [My Great Products]
FROM Products;
```

- Using "double quotes" for aliases with space characters:

```
SELECT ProductName AS "My Great Products"
FROM Products;
```

```
select ename, sal, sal*12 annual
from emp;
```

- "" (double quotes) is optional but If you give value in LOWERCASE, output will come in UPPERCASE
- By default it comes in UPPERCASE
- So it is compulsory to give value in ""(double quotes) to make it case sensitive

```
select ename, sal, sal*12 "Annual Salary in $"
from emp;
```

```
select ename "EMPNAME",
sal "SALARY",
sal*12 "ANNUAL",
sal*12*0.4 "HRA",
sal*12*12 "DA",
sal*12 + sal*12*0.4 + sal*12*0.2 "NET"
from emp;
```

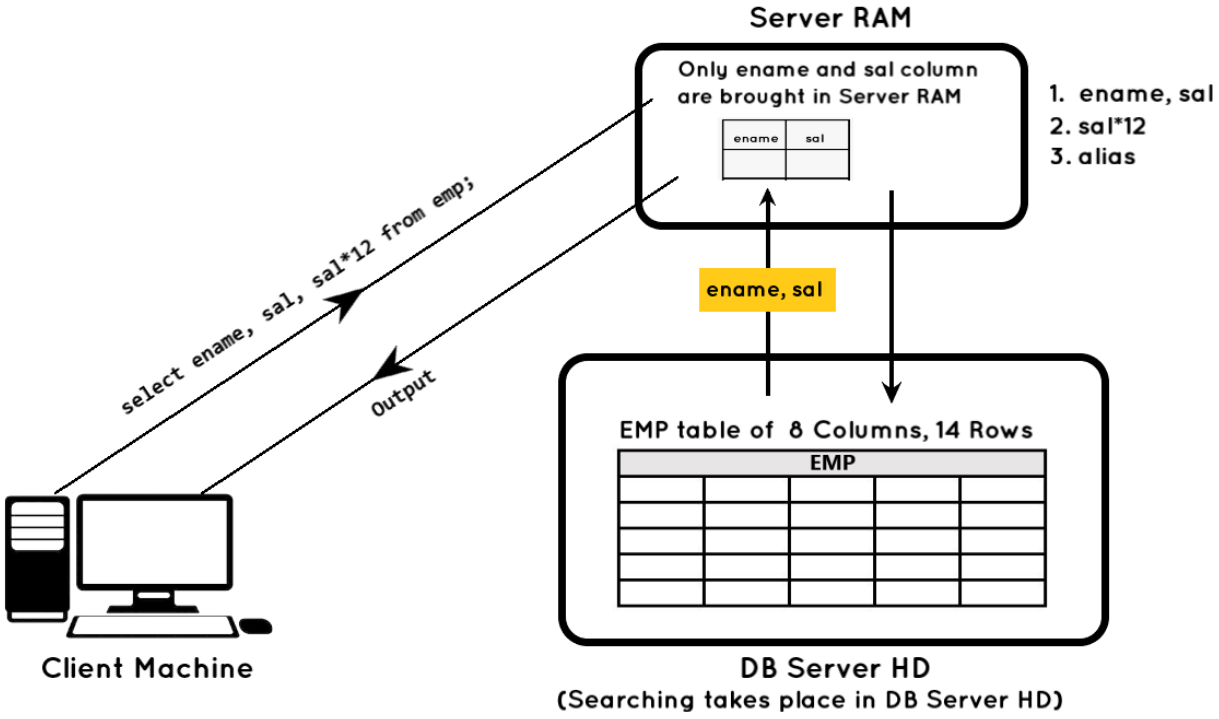
- Alias cannot be used in expressions

- Alias cannot be used in WHERE clause
  (because WHERE clause is used for searching, searching takes place in DB server HD, at the time of searching inside the table annual column does not exist, annual is calculated by MySQL after rows are brought in Server RAM)

```
select ename "EMPNAME",
sal "SALARY",
sal*12 "ANNUAL"
from emp
where annual > 300000;  ---------(ERROR)
```

- Alias can be used with expression in WHERE clause

```
select ename "EMPNAME",
sal "SALARY",
sal*12 "ANNUAL"
from emp
where sal*12  > 300000;  --------(OK)
```



DISTINCT

- The SELECT DISTINCT statement is used to return only distinct (different) values.
- MySQL DISTINCT clause is used to remove duplicate records from the table and fetch only the unique records.
- The DISTINCT clause is only used with the SELECT statement.

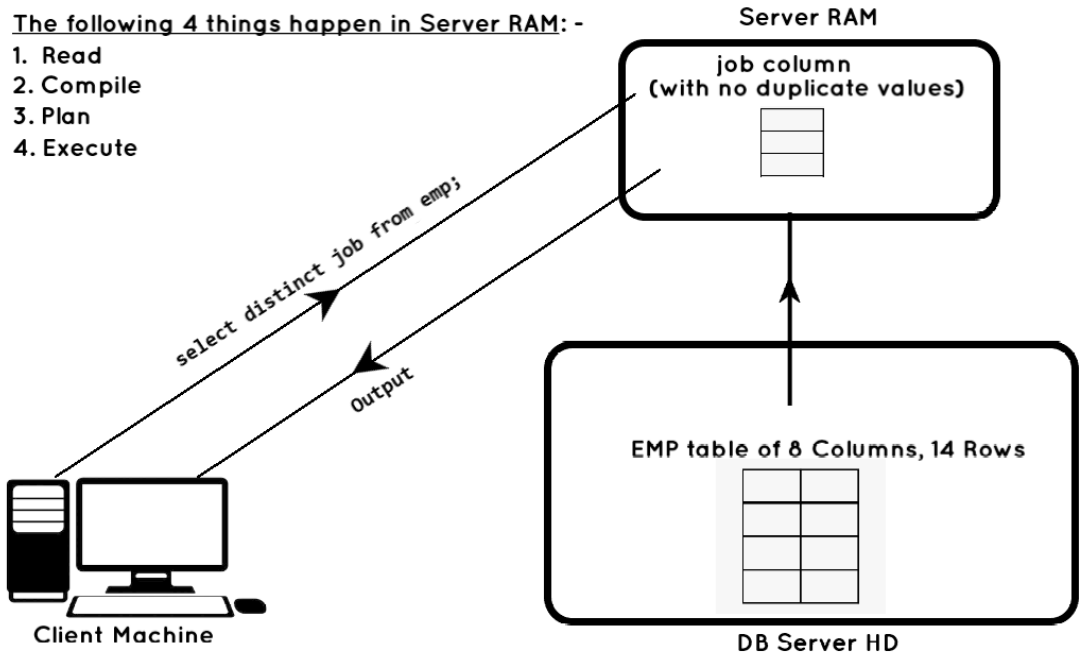
To suppress the duplicate JOBS: -

```
select distinct job from emp;
```

- When you use DISTINCT, sorting takes place in Server RAM
- Sorting is one of operation that slows down the SELECT statement;
  if you have a large number of rows in the table, then it will slow

To suppress the duplicate JOBS: -

```
select distinct job from emp;
```




```
select distinct job, ename from emp;
```

ERROR

```
select (distinct job), ename from emp;
```

Installation of MySQL

When you install MySQL, 2 users are automatically created

1. Root (password: cdac)
2. mysql.sys

1) Root
   - DBA privileges
   - create database, drop database, alter database, configure database, create users, assign privileges, take backups, performance planning, performance monitoring, performance tuning, performance management, etc.
2) mysql.sys
   - most important user in MySQL
   - owner of database and system tables
   - start-up database, shutdown database, perform Recovery

1. Login with root user

cmd > mysql -u root -p

Shortcut for MySQL Command Line Client

Start Menu→search MySQL Command Line Client icon and select→Right mouse Click→ Open File Location→ select MySQL Command Line Client →Right mouse click→Properties

Target -> "C:\Program Files\MySQL\MySQL Server 8.0\bin\mysql.exe" "--defaults-file=C:\ProgramData\MySQL\MySQL Server 8.0\my.ini" "-uroot" "-p"

"C:\Program Files\MySQL\MySQL Server 8.0\bin\mysql.exe" "--defaults-file=C:\ProgramData\MySQL\MySQL Server 8.0\my.ini" "-uroot" "-pgdac"

```
mysql>show databases;
mysql>use <database name>;
mysql>use mysql
mysql>select * from user;
```

user is a system table (it contain all usernames)

How to create new user?

(With, username – prasad; password – student; database name - juhu**)**

Step 1: To create a new user

```
mysql>create user <user name> identified by <password>;
mysql>create user prasad@localhost identified by 'student';
```

Step 2: To create a new database / schema

schema is synonym for database

```
mysql>create database juhu;
OR
mysql>create schema juhu;
mysql>show databases;
```

Step 3: To grant all permission to the new user on the new database;

```
mysql>grant all privileges on juhu.* to prasad@localhost;
mysql>flush privileges;
```

```
mysql>flush privileges;
```

reloads the grant tables in the MySQL database enabling the change to take effect without reloading or restarting the MySQL database/service

```
mysql>select user from user;
mysql>exit;
```

Shortcut for MySQL Command Line Client

```
cmd>mysql -u prasad -p
Password: student
```

```
mysql>show databases;
mysql>use juhu;
mysql>show tables;
mysql>desc emp;                → describes emp table
```

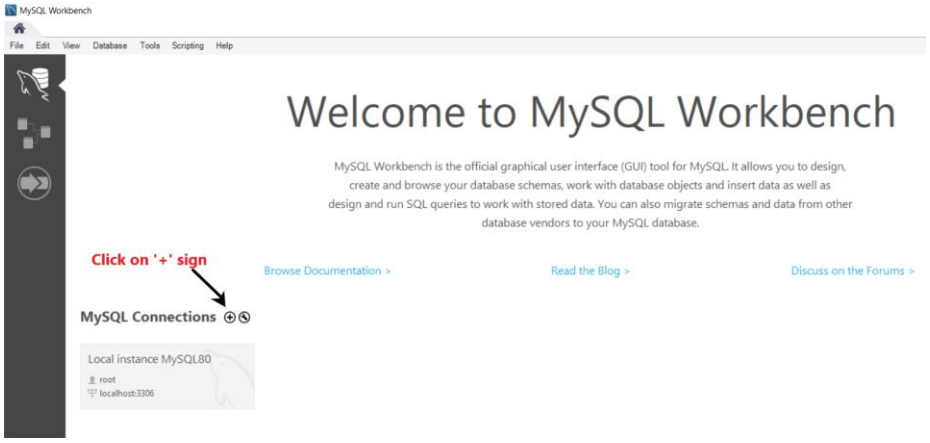
DAY 3

Working with MySQL Workbench

Start→ MySQL Workbench→

1. To connect to MySQL database using root user: -

- MySQL Connections (Click on (+) sign to create a new connection)



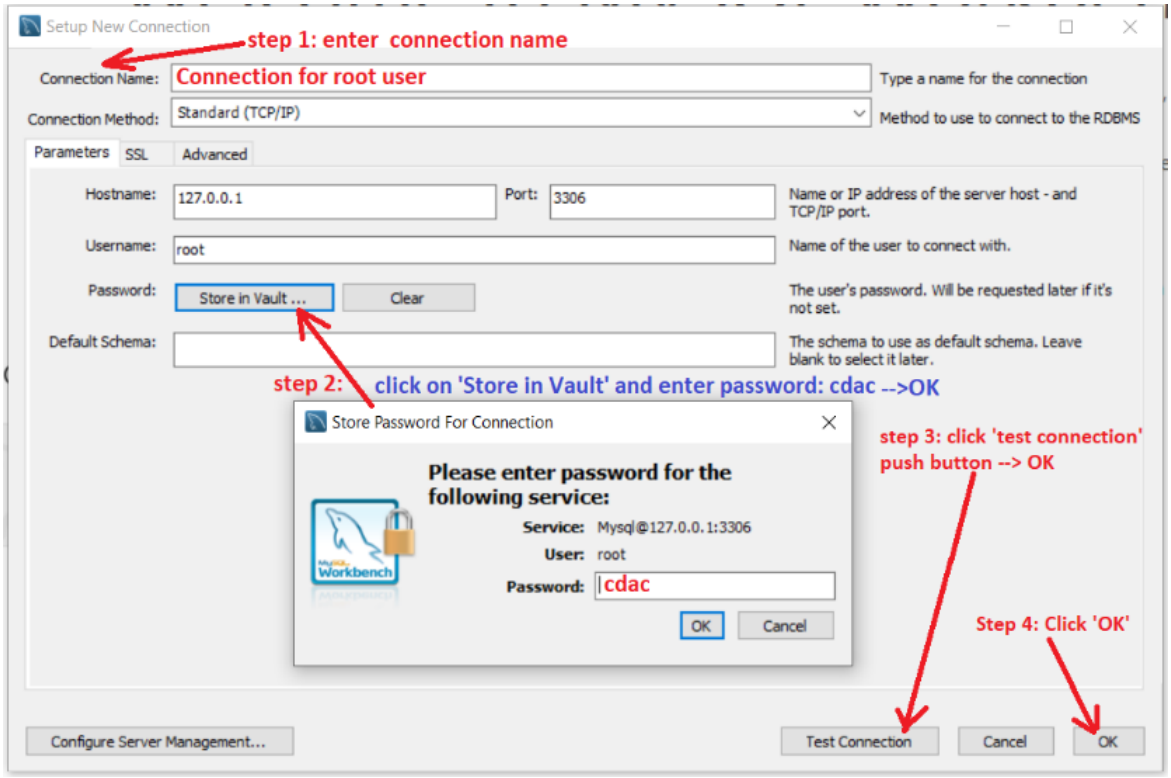
- Connection name: - **Connection for root user**
- Connection method: - **Standard (TCP/IP)** ← network protocol
- Hostname: - cdac.in or 192.168.0.1 or **localhost**
- Port: - **3306**
  (Default port number for MySQL: - 3306 and Oracle: - 1521)
- Username: - **root**
- Password: - **cdac**
  ((Store in vault … Push button) → Click on it and enter the password → cdac)
- Default Schema: -
  (leave it blank)

Test Connection(push button)→ Click on it

- OK (push button)→ Click on it

OK (push button)→ Click on it

'We have made connection for root user'



2. Click on the connection you created i.e., **Connection for root user (MySQL workbench will open)**
   - You will see Object Navigator on left hand side
   - You will see Query window at the top
   - You will see Output window below

Some basic commands post login: -

```
show databases;  /*→ Press Ctrl + enter to execute*/
```

```
use mysql;
select * from user;
select user from user;
```

```
create database juhu; /*→ it will create database juhu*/
```

```
show databases; /*→ you can see juhu database in database list*/
```

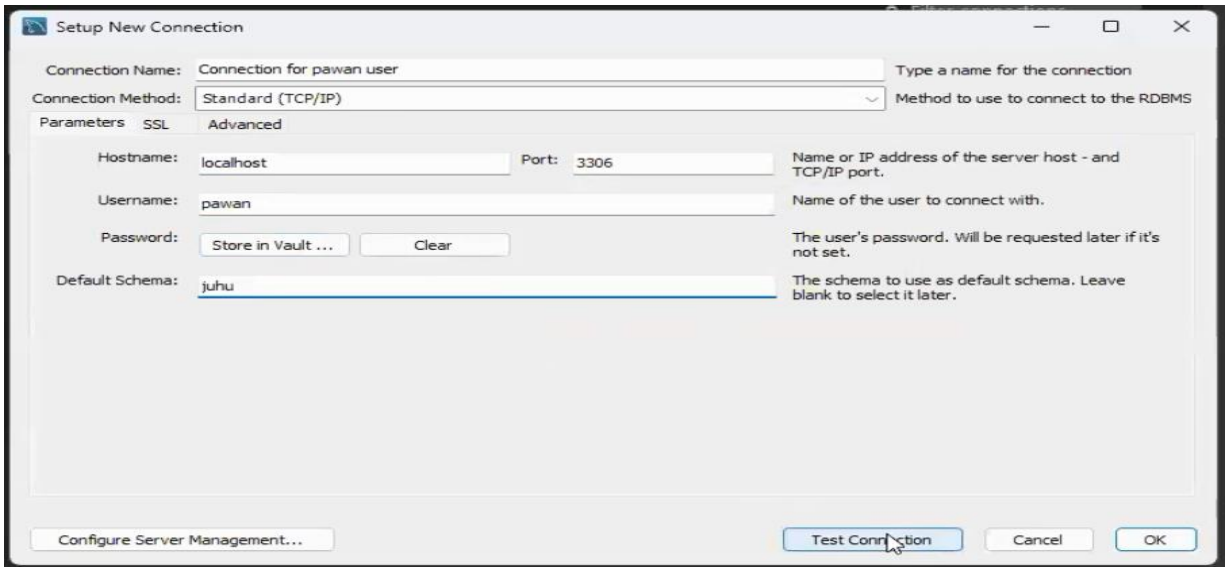
```
create user pawan@'%' identified by 'student';
```

- @'%' means pawan user can connect to database from any computer in the world
- student is password

To grant the permissions: -

- Click on Server (menu at the top)
- Users and Privileges (Click on it)
- Select the username you created from the User Accounts List on LHS
- (Tab menu) → administrative roles→ Click on it
- DBA (checkbox) → Click on it
- Apply (push button)→ Click on it
- Schema Privileges (tab)→ Click on it
- Add Entry…..(push button)→ Click on it
- Selected Schema (radio button)→ Click on it
- (Pop list)→ Select juhu)→ Click on OK
- Select "All" (push button)→ Click on it
  (Note when you click on Select "All", grant option checkbox is unchecked)
- Grant Option (checkbox)→ Click on it
- Apply (push button)→ Click on it

3. Exit from MSQL Workbench
4. Create a new connection for pawan user (Default schema: - juhu) → Click on OK → Click on OK→Yahoo user is created you can start working



```
select deptno, job, ename, sal, hiredate from emp;
```

- In a DBMS, the rows are stored inside a file
- Inside a file, the data is stored sequentially
- In RDBMS, table is not a file; every row is a file
- Rows inside the table are not stored sequentially.
- Rows inside the table are scattered (fragmented) all over the DB server HD
- The reason RDBMS does this is to speed up the INSERT statement (from a multi-user perspective)
- In a multi-user environment, if a large number of users are inserting rows in the same table at the same time, if MySQL were to the rows sequentially it would be very slow; wherever it finds the free HD space it will store the row there
- Once you INSERT a row, the row address is constant
- When you UPDATE the row, if the length is increasing, and if the free space is not available, then the entire row is moved to some other address
- hence it is not possible to see the first 'N' rows or the last 'N' rows of a table
- If the row length decreasing after UPDATE, then the row address will not change
- It's only in the case of varchar that row length may increase or decrease

ORDER BY clause

- It is used for sorting (sorting takes place in server RAM)

```
select deptno, job, ename, sal, hiredate from emp
order by ename;
```

- it will sort rows of table on basis of ename column

```
select deptno, job, ename, sal, hiredate from emp
order by ename desc;
```

asc → for ascending → by default
desc → for descending

```
select deptno, job, ename, sal, hiredate from emp
order by deptno;
```

- row sorting takes place according to deptno in ascending order

Uses: - used for presentation purpose (for reporting purposes)

```
select deptno, job, ename, sal, hiredate from emp
order by deptno;
```

- Row order is coming according to deptno column in ascending order

- BUSINESS INTELLIGENCE
- DATA ANALYTICS
- MACHINE LEARNING
- DATA SCIENCE
- ARTIFICIAL INTELLIGENCE

```
select deptno, job, ename, sal, hiredate from emp
order by deptno, job;
```

- first it will sort on basis of deptno, if deptno is same then it will sort on basis of job and so on…….

```
select deptno, job, ename, sal, hiredate from emp
order by deptno desc, job;
```

```
select deptno, job, ename, sal, hiredate from emp
order by deptno desc, job desc;
```

- There is no upper limit on the number of columns in ORDER BY clause

select……
order by country, state, city;

- Sorting is an operation that slows down your SELECT statement
- If you have large number of rows in the table, and if you have large number of columns in ORDER BY clause, then SELECT statement will be very slow (because that much sorting takes place in Server RAM)

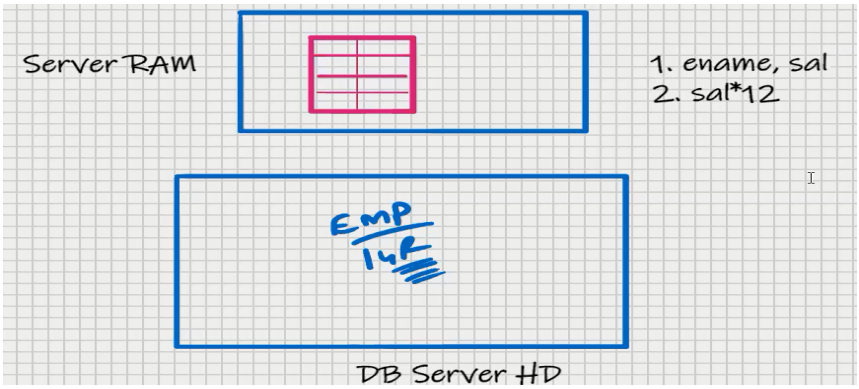
```
select deptno, job, ename, sal, hiredate from emp
where deptno = 10
order by ename;
```

- WHERE clause is specified BEFORE the ORDER BY clause
- WHERE clause is used for searching
- Searching takes place in DB Server HD
- WHERE clause is used to restrict the rows
- WHERE clause is used to retrieve the rows from DB Server HD to Server RAM
- ORDER BY clause sorting takes place after the rows are brought into the Server RAM
- ORDER BY clause is the last clause in the SELECT statement

```
select ename, sal*12 from emp;
```

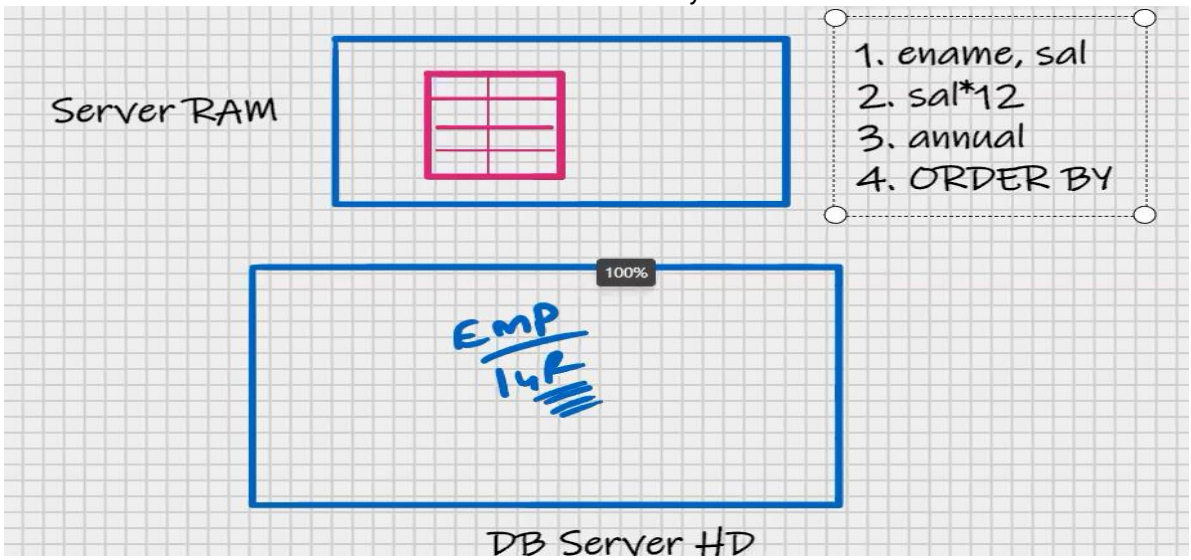
```
select ename, sal*12 from emp
order by sal*12;
```

- In ORDER BY expression is allowed
- ename, sal column is brought in server RAM, MySQL will create an 2D array which contains ename column and sal column and then it will go inside FOR loop. It will calculate respective totals i.e., sal*12 and then ORDER BY sorting will takes place




```
select ename, sal*12 annual from emp
order by annual;
```

- You can use ALIAS in ORDER BY clause but you cannot use ALIAS in WHERE clause




```
select ename, sal*12 "Annual Salary" from emp
order by "Annual Salary";
```

```
select ename, sal*12 "Annual Salary" from emp
order by 2;
```

- Here 2 is column number, means sort the second column of table

```
select * from emp
order by 2;
```

EMP

EMPNO	ENAME	SAL	CITY	DEPTNO
1	ADAMS	1000	Mumbai	10
2	BLAKE	2000	Delhi	10
3	ALLEN	2500	Mumbai	20
4	KING	3000	Delhi	25
5	FORD	4000	Mumbai	30

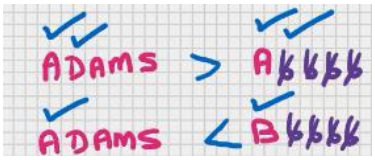
```
create table emp
(
EMPNO char(4),
ENAME varchar(25),
SAL float,
CITY varchar(15),
DEPTNO int
);
```

```
insert into emp(empno, ename, sal, city, deptno)
values('1', 'ADAMS', 1000, 'Mumbai', 10),
('2', 'BLAKE', 2000, 'Delhi', 10),
('3', 'ALLEN', 2500, 'Mumbai', 20),
('4', 'KING', 3000, 'Delhi', 25),
('5', 'FORD', 4000, 'Mumbai', 30);
```

Display all the rows starting with 'A': -

```
select * from emp
where ename > 'A' and ename < 'B';
```

	EMPNO	ENAME	SAL	CITY	DEPTNO
▶	1	ADAMS	1000	Mumbai	10
	3	ALLEN	2500	Mumbai	20



Blank-Padded comparison semantics:

When you compare 2 strings of different lengths, the shorter of the 2 strings is temporarily padded with blank spaces on RHS such that their lengths become equal, then it will start the comparison character by character based on ASCII value
Note: In case while comparing ASCII values, if both the letter are same then comparison will be of ASCII values next adjacent letters
e.g., comparing 'ADAMS' with 'A'
here first letter 'A' is same for both, so comparison of ASCII values of next letter D and ASCII values of blank space will take place

```
select * from emp
where ename >= 'A' and ename < 'B';
```

```
select * from emp
where ename >= 'A' and ename < 'C';
```

1) Using the LIKE operator with the % wildcard examples

```
SELECT
    customer_id,
    first_name,
    last_name
FROM
    sales.customers
WHERE
    last_name LIKE 'z%'
ORDER BY
    first_name;
```

The following example uses the LIKE operator with the % wildcard to find the customers whose last name starts with the letter z:

customer_id	first_name	last_name
1354	Alexandria	Zamora
304	Jayme	Zamora
110	Ollie	Zimmerman

```
SELECT
    customer_id,
    first_name,
    last_name
FROM
    sales.customers
WHERE
    last_name LIKE '%er'
ORDER BY
    first_name;
```

The following example uses the LIKE operator with the % wildcard to return the customers whose last name ends with the string er:

customer_id	first_name	last_name
1412	Adrien	Hunter
62	Alica	Hunter
619	Ana	Palmer
525	Andreas	Mayer
528	Angele	Schroeder
1345	Arie	Hunter
851	Arlena	Buckner
477	Arminda	Weber
425	Augustina	Joyner
290	Barry	Buckner
1169	Beatris	Joyner

```
SELECT
    customer_id,
    first_name,
    last_name
FROM
    sales.customers
WHERE
    last_name LIKE 't%s'
ORDER BY
    first_name;
```

The following statement uses the LIKE operator to retrieve the customers whose last name starts with the letter t and ends with the letter s:

customer_id	first_name	last_name
682	Amita	Thomas
904	Jana	Thomas
1360	Latashia	Travis
567	Sheila	Travis

To retrieve data from a table, you use the SELECT statement with the following syntax:
SELECT select_list FROM schema_name.table_name;
In this syntax:
First, specify a list of comma-separated columns from which you want to query data in the SELECT clause.
Second, specify the table name and its schema in the FROM clause.

When processing the SELECT statement, SQL Server first processes the FROM clause, followed by the SELECT clause, even though the SELECT clause appears before the FROM clause:

2) Using the LIKE operator with the _ (underscore) wildcard example

```
SELECT
    customer_id,
    first_name,
    last_name
FROM
    sales.customers
WHERE
    last_name LIKE
'_u%'
ORDER BY
    first_name;
```

The underscore represents a single character. For example, the following statement returns the customers where the second character is the letter u:

customer_id	first_name	last_name
338	Abbey	Pugh
1412	Adrien	Hunter
527	Afton	Juarez
442	Alane	Munoz
62	Alica	Hunter
683	Amparo	Burks
1350	Annett	Rush
1345	Arie	Hunter
851	Arlena	Buckner
1200	Aubrey	Durham
290	Barry	Buckner

3) Using the LIKE operator with the [list of characters] wildcard example

```
SELECT
    customer_id,
    first_name,
    last_name
FROM
    sales.customers
WHERE
    last_name LIKE
'[YZ]%'
ORDER BY
    last_name;
```

The square brackets with a list of characters
e.g., [ABC] represents a single character that must be one of the characters specified in the list.
For example, the following query returns the customers where the first character in the last name is Y or Z:

customer_id	first_name	last_name
54	Fran	Yang
250	Ivonne	Yang
768	Yvone	Yates
223	Scarlet	Yates
498	Edda	Young
543	Jasmin	Young
1354	Alexandria	Zamora
304	Jayme	Zamora
110	Ollie	Zimmerman

4) Using the LIKE operator with the [character-character] wildcard example

```
SELECT customer_id, first_name, last_name
FROM sales.customers
WHERE last_name LIKE '[A-C]%'
ORDER BY first_name;
```

The square brackets with a character range
e.g., [A-C] represent a single character that must be within a specified range.

For example, the following query finds the customers where the first character in the last name is the letter in the range A through C:

customer_id	first_name	last_name
1224	Abram	Copeland
1023	Adena	Blake
1061	Alanna	Barry
1219	Alden	Atkinson
1135	Alisia	Albert
892	Alissa	Craft
1288	Allie	Conley
1295	Alline	Beasley
1168	Almeta	Benjamin
683	Amparo	Burks
947	Angele	Castro

```
SELECT name FROM
sys.databases
WHERE name LIKE 'm[n-
z]%';
```

The following example returns names that start with the letter m. [n-z] specifies that the second letter must be somewhere in the range from n to z. The percent wildcard % allows any or no characters starting with the third character.

5) Using the LIKE operator with the [^Character List or Range] wildcard example

```
SELECT
    customer_id,
    first_name,
    last_name
FROM
    sales.customers
WHERE
    last_name LIKE
'[^A-X]%'
ORDER BY
    last_name;
```

The square brackets with a caret sign (^) followed by a range e.g., [^A-C] or character list e.g., [ABC] represent a single character that is not in the specified range or character list.
For example, the following query returns the customers where the first character in the last name is not the letter in the range A through X:

customer_id	first_name	last_name
54	Fran	Yang
250	Ivonne	Yang
768	Yvone	Yates
223	Scarlet	Yates
498	Edda	Young
543	Jasmin	Young
1354	Alexandria	Zamora
304	Jayme	Zamora
110	Ollie	Zimmerman

6) Using the NOT LIKE operator example

```
SELECT
    customer_id,
    first_name,
    last_name
FROM
    sales.customers
WHERE
    first_name NOT LIKE
'A%'
ORDER BY
    first_name;
```

The following example uses the NOT LIKE operator to find customers where the first character in the first name is not the letter A:

customer_id	first_name	last_name
174	Babara	Ochoa
1108	Bao	Wade
225	Barbera	Riggs
1249	Barbra	Dickerson
802	Barrett	Sanders
1154	Barry	Albert
290	Barry	Buckner
399	Bart	Hess
269	Barton	Crosby
977	Barton	Cox

MySQL–SQL-Special Operators (Like, Between, Any, In):

Wildcard (used for pattern matching)

- % any character and any number of characters
- _ any 1 character

	Commands	**Meaning**
1	`select * from emp where ename like 'A%';`	You will see the names starting with letter 'A'
2	`select * from emp where ename = 'A%';`	It will search for A%
3	`select * from emp where ename like '%A';`	You will see the names ending with letter 'A'
4	`select * from emp where ename like '%A%';`	You will see all the rows that contains the letter A i.e., starting with 'A' **or** ending with 'A' **or** 'A' in the middle
5	`select * from emp where ename like 'DAC%';`	You will see the names starting with 'DAC'
6	`select * from emp where ename like '_ _A%';`	You will see the names that contains 'A' as fourth letter/character (Note: Blank space is counted)
7	`select * from emp where ename like 'A%A';`	Starting with letter 'A' and ending with letter 'A'
8	`select * from emp where ename like '_ _ _ _';`	You will see all the names with 4 letters
9	`select * from emp where ename not like 'A%';`	You will see the names **not** starting with letter 'A'

<table>
<tr><td>

```
select * from emp
where sal >= 2000 and sal <= 3000;
```

</td><td rowspan="2">Inclusive</td></tr>
<tr><td>

```
select * from emp
where sal > 2000 and sal < 3000;
```

</td></tr>
<tr><td>

```
select * from emp
where sal not between 2000 and 3000;
```

</td><td rowspan="8">Exclusive</td></tr>
<tr><td>

```
select * from emp
where hiredate between '2022-01-01' and '2022-12-31';
```

</td></tr>
<tr><td>

```
select * from emp
where hiredate >= '2022-01-01' and hiredate <= '2022-12-31';
```

</td></tr>
<tr><td>

```
select * from emp
where hiredate >= '2022-01-01' and hiredate <= '2022-12-31';
```

</td></tr>
<tr><td>

```
select * from emp
where ename between 'A' and 'B';
```

- ename starting with letter 'A'
  greater than equal to (> =) A and less than equal to (<=) 'B'

```
select * from emp
where ename >= 'A' and ename <= 'B';
```

</td></tr>
<tr><td>

select * from emp
where ename between 'A' and 'B' and ename != 'B';

</td></tr>
<tr><td>

```
SELECT * FROM emp
WHERE ename BETWEEN 'A' AND 'E';
```

- 'A' and 'E' are treated as strings, so it compares names alphabetically.
- 'E' does not include names starting with 'E' followed by any letter (e.g., 'Edward'), because 'Edward' > 'E' in string comparison.
- It includes values like 'A', 'B', 'C', 'D', and **only 'E' exactly**, not 'E.....'.

To match all names starting with A through E (more naturally):
If your goal is to select all employees whose names start with A, B, C, D, or E, then the BETWEEN won't give you what you expect.

```
SELECT * FROM emp
WHERE ename >= 'A' AND ename < 'F';
```

NOTE:-

```
SELECT * FROM emp
WHERE ename >= 'A' AND ename <= 'E';
```

- **Yes, it will run without errors, but it likely won't behave the way you expect. It includes values like 'A', 'B', 'C', 'D', and only 'E' exactly, not 'E.....'.**

</td></tr>
</table>

```
select * from emp
where deptno = 10 or deptno = 20 or deptno = 30;
```

```
select * from emp
where deptno =any(10,20,30);       ----------→(Logical OR)
```

```
select * from emp
where deptno in(10,20,30);          ----------→(Logical OR)
```

```
select * from emp
where deptno not in(10,20,30);
```

```
select * from emp
where deptno =any(10,20,30);
```

```
select * from emp
where deptno !=any(10,20,30);
```

```
select * from emp
where deptno >any(10,20,30);
```

```
select * from emp
where deptno >=any(10,20,30);
```

```
select * from emp
where deptno <any(10,20,30);
```

```
select * from emp
where deptno <=any(10,20,30);
```

- IN operator works faster than ANY operator
- ANY operator is more powerful than IN operator
- With the IN operator, you can check for IN and NOT IN
- With the ANY operator, you can check for =ANY, !=ANY, >ANY, >=ANY, <ANY, <=ANY
- If you want to check for equality or inequality, then use the IN operator
- If you want to check for >, >=, <, <= then use the ANY operator
- If the requirement is more complex, then use Relational and Logical operators

```
select * from emp
where city in('Mumbai', 'Delhi');
```

- IN operator works in all RDBMS including MySQL and Oracle
- ANY operator works in Oracle RDBMS
- ANY operator works in MySQL RDBMS only if it is used in sub- query

UPDATE: Used to modify existing data.

Syntax: -
update *table_name*
set *column_name* = *new_value*
where *condition*;

```
update emp
set sal = 10000
where empno = 1;
```

EMP

EMPNO	ENAME	SAL	CITY	DEPTNO
1	ADAMS	1000 → 10000	Mumbai	10
2	BLAKE	2000	Delhi	10
3	ALLEN	2500	Mumbai	20
4	KING	3000	Delhi	25
5	FORD	4000	Mumbai	30

```
update emp
set sal = 10000, city = 'Pune'
where empno = 1;
```

- It will not only update existing sal to 10000 but also update the city to Pune in row 1

```
update emp
set sal = sal + sal*0.4
where empno = 1;
```

```
update emp
set sal = 10000
where city = 'Mumbai';
```

- It will update sal to 10000 of all those employees whose city is Mumbai

```
update emp
set sal = 10000;
```

- It will update all the rows of 'sal' column to 10000
- Be carefull while updating records. **If you omit WHERE clause , all records will be updated**

```
update emp
set sal = 10000, city = 'Pune'
where city = 'Mumbai';
```

- It will update all the row of employees whose city is 'Mumbai' to 'Pune' and sal to 10000

EMP

EMPNO	ENAME	SAL	CITY	DEPTNO
1	ADAMS	1000 → 10000	Mumbai → Pune	10
2	BLAKE	2000	Delhi	10
3	ALLEN	2500 → 10000	Mumbai → Pune	20
4	KING	3000	Delhi	25
5	FORD	4000 → 10000	Mumbai → Pune	30

- You can UPDATE multiple rows and multiple columns simultaneously, but you can UPDATE only 1 table at a time
- If you want to UPDATE multiple tables, then a separate UPDATE command is required for each table

DELETE: It is used to delete existing records in the table

DELETE specific rows from the table

```
delete from emp
where empno = 1;
```

- The complete command deletes all rows from the `emp` table where the `empno` column has a value of `1`. If there is only one row with `empno = 1`, only that row will be deleted. If there are multiple rows with `empno = 1`, all of those rows will be deleted.

```
delete from emp
where city = 'Mumbai';
```

- The complete command deletes all rows from the `emp` table where the `city` column has a value of `'Mumbai'`. If there are multiple rows with `city = 'Mumbai'`, all of those rows will be deleted.

DELETE all rows from the table

```
delete from emp;
```

- Since there is no `WHERE` clause specified, this command will delete **all rows** from the `emp` table.
- All the rows will be deleted, but complete table will not get deleted, empty table and column will remain

DROP

To delete complete table: -	
`drop table emp;`	Entire table will be deleted completely

To delete multiple table: -

```
drop table emp, dept;
```

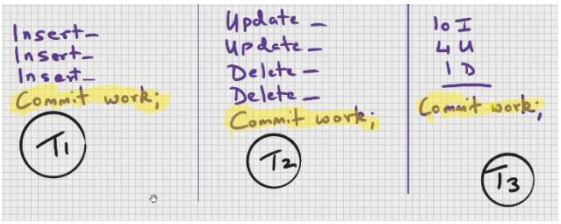
- You cannot use WHERE clause with DROP table, because it is a DDL command
- If you want to drop multiple tables, then you will have to drop each table separately
- A separate DROP table command would be required for each table

TRANSACTION PROCESSING

COMMIT

- Commit will save all the DML (insert, update, delete) changes since the last committed state
- When the user issues a COMMIT, it is known as End of Transaction
- COMMIT will make transaction permanent
- Transaction is unit of Work

Total Work done = T1+T2+T3+……+Tn



- When to issue the COMMIT is decided by the client (end user) and it depends on the logical scope of Work

```
Commit work;
or
Commit;
```

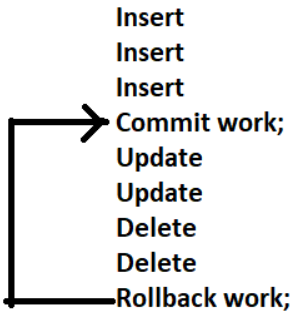
- WORK is optional in MySQL and Oracle
- WORK is ANSI SQL

ROLLBACK

- Rollback will undo all the DML changes since the last committed state

```
rollback work;
or
rollback;
```

- WORK is optional in MySQL and Oracle
- WORK is ANSI SQL



- Any DDL command, it automatically commits
- You cannot Commit or Rollback the DML changes made in other sessions
- When you exit from SQL, it automatically commits
- Any kind of power failure, network failure, system failure, PC reboot, window close, improper exit from SQL, etc.; your last uncommitted Transaction is automatically Rolled back

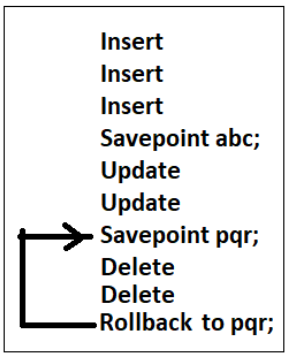
```
mysql>set autocommit = 1;      OR   set autocommit = ON    (by default)

mysql>set autocommit = 0;      OR   set autocommit = OFF
```

- A COMMIT means that the changes made in the current transaction are made permanent and become visible to other sessions. A ROLLBACK statement, on the other hand, cancels all modifications made by the current transaction.
- Both COMMIT and ROLLBACK release all InnoDB locks that were set during the current transaction. By default, connection to the MySQL server begins with autocommit mode enabled, which automatically commits every SQL statement as you execute it. This mode of operation might be unfamiliar if you have experience with other database systems, where it is standard practice to issue a sequence of DML statements and commit them or roll them back all together.
- To use multiple-statement transactions, switch autocommit off with the SQL statement SET autocommit = 0 and end each transaction with COMMIT or ROLLBACK as appropriate. To leave autocommit on, begin each transaction with START TRANSACTION and end it with COMMIT or ROLLBACK. The following example shows two transactions. The first is committed; the second is rolled back.

```
mysql> CREATE TABLE customer (a INT, b CHAR (20), INDEX (a));
Query OK, 0 rows affected (0.00 sec)
mysql> -- Do a transaction with autocommit turned on.
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO customer VALUES (10, 'Heikki');
Query OK, 1 row affected (0.00 sec)
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
mysql> -- Do another transaction with autocommit turned off.
mysql> SET autocommit=0;
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO customer VALUES (15, 'John');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO customer VALUES (20, 'Paul');
Query OK, 1 row affected (0.00 sec)
mysql> DELETE FROM customer WHERE b = 'Heikki';
Query OK, 1 row affected (0.00 sec)
mysql> -- Now we undo those last 2 inserts and the delete.
mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT * FROM customer;
+------+--------+
| a    | b      |
+------+--------+
|   10 | Heikki |
+------+--------+
1 row in set (0.00 sec)
mysql>
```

SAVEPOINT: -



- You can Rollback to a savepoint
- Savepoint is a point within a work
- Savepoint is similar to a Bookmark/Flag/Milestone
- Savepoint is a sub-unit of Transaction
- You can Rollback to a Savepoint
- YOU CANNOT COMMIT TO A SAVEPOINT
- Commit will save all the DML (insert, update, delete) changes since the last committed state
- You can only Rollback sequentially
- When you Rollback or Commit, the intermediate Savepoints are automatically cleared; if you want to use them again, then you will have to reissue them in some new Work
- Within a Transaction, you can have 2 Savepoints with the same name; the latest Savepoint supersedes the older one; the older Savepoint no longer exists

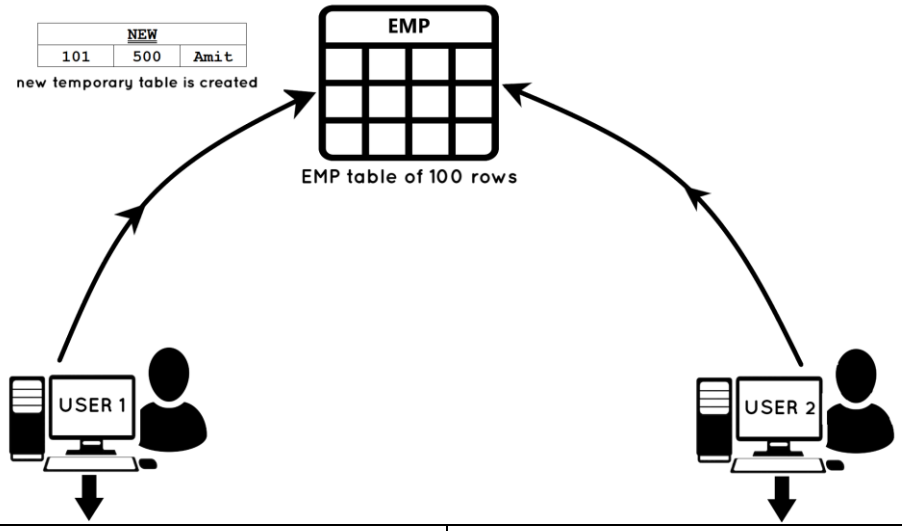
DAY 4

MySQL – SQL – Read and Write consistency

- When you SELECT a table, you can view only the committed data of all users/sessions Plus
  the changes made by you
- When you UPDATE or DELETE a row, that row is automatically locked for other users
- Row locking in MySQL and Oracle is automatic
- When you UPDATE or DELETE a row, that row becomes READ_ONLY for other users
- Other users can SELECT from that table; they will view the old data before your changes
- Other users can INSERT into that table
- Other users can UPDATE or DELETE "other" rows
- No other user can UPDATE or DELETE your locked row till you have issued a Rollback or Commit
- LOCKS ARE AUTOMATICALLY RELEASED WHEN YOU ROLLBACK OR COMMIT

What happens when user issues INSERT command ?

Suppose you have an `emp` table with 100 rows and a multi-user environment involving User 1 and User 2. Both the users are accessing `emp` table



`select * from emp;` • USER 1 can see 100 rows in the emp table	`select * from emp;` • USER 2 can see 100 rows in the emp table
`insert into emp values(101,'Amit',5000)`	
`select * from emp;` • USER 1 can see 101 rows in the emp table, because new inserted row (101,'Amit',5000) is updated in emp table	`select * from emp;` • USER 2 can see 100 rows in the emp table, newly inserted row is not visible(updated) for USER 2
`commit;`	
	`select * from emp;` • USER 2 can now see 101 rows in the emp table, newly inserted row is visible(updated) for USER 2 when USER 1 gives COMMIT
When multiple users are accessing or modifying the `emp` table simultaneously, database systems use locking mechanisms to ensure data consistency. For instance, if user 1 is updating a row, user 2 might be blocked from accessing that row until the update is complete.	

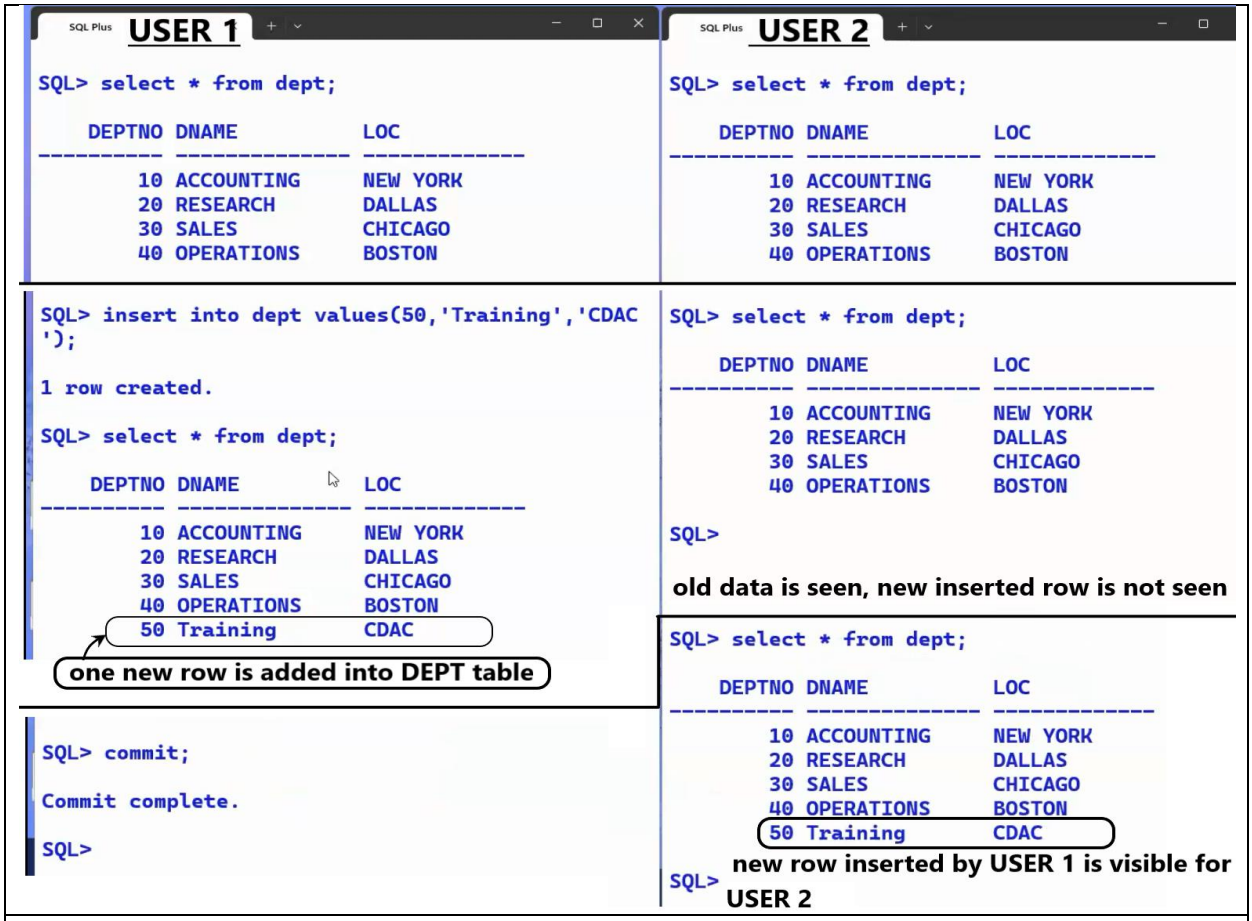
- When USER 1 decides to insert a row (101, 'Amit', 5000)

This newly inserted row (101, 'Amit', 5000) has gone to the database, this one row is there on the server, this row is there inside the hard disk, but this new row inserted is not copied inside the main EMP table.

Instead MySQL creates temporary table of name NEW and it stores a new row there. This temporary table is user-specific, meaning each user has their own version of the NEW table (There would be a separate NEW temporary table for every user)

- When USER 1 runs `SELECT * FROM emp`, MySQL shows the data from the `EMP` table along with the row((101, 'Amit', 5000)) in the temporary `NEW` table for USER 1's session. Therefore, USER 1 sees 101 rows.
- Meanwhile, if USER 2 runs `SELECT * FROM emp`, MySQL does not consider the temporary `NEW` table created by USER 1, so USER 2 sees only the original 100 rows in the `EMP` table.

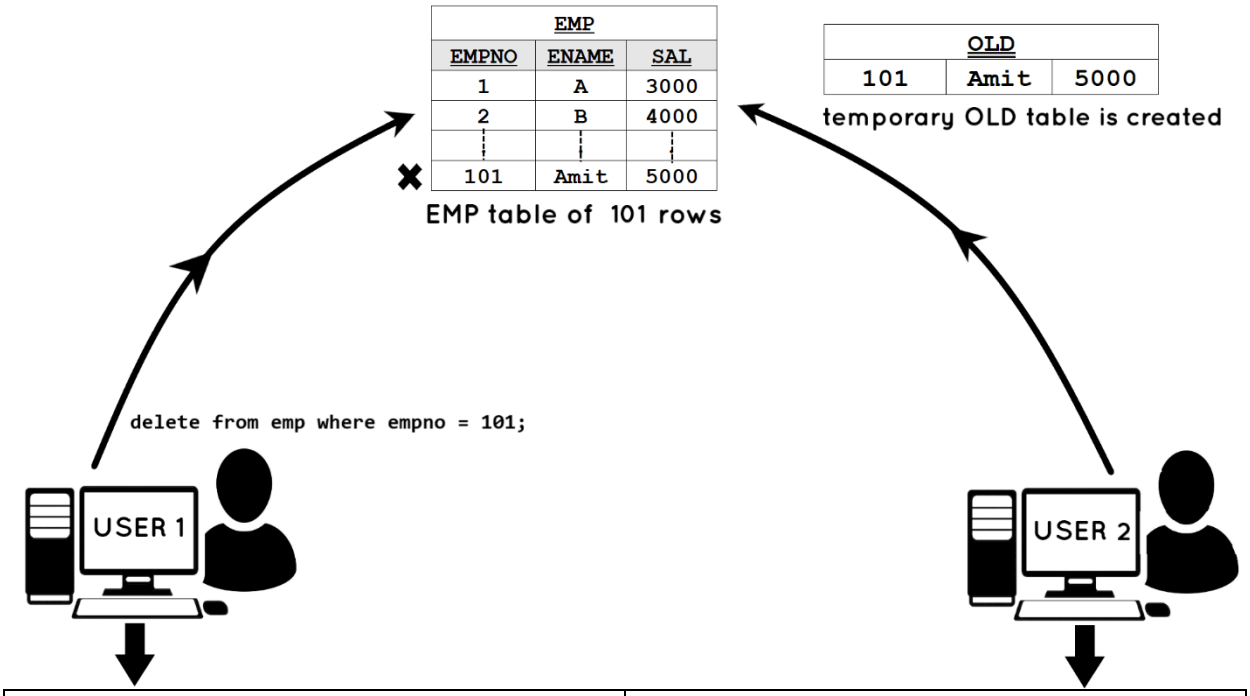
- When USER 1 issues a `COMMIT` command, the row in the temporary `NEW` table is permanently added to the `EMP` table.
- The temporary table `NEW` for USER 1 is then dropped, and the changes become visible to all users.
- After issuing `COMMIT` command by USER 1,
  when USER 2 performs `SELECT * FROM emp`, USER 2 will now see 101 rows, reflecting the committed changes made by USER 1.



- **USER 1 inserted a new row but inserted new row is not visible for USER 2.**
- **USER 2 can see changes made by USER 1 only when USER 1 gives commit.**
- **When USER 1 gives commit, table becomes permanent, but what internally happens is that inserted new row is added to dept table, and new temporary table will be droped after which USER 2 can see new inserted row.**
- **When user 1 gives commit this new row is inserted into EMP table and temporary table is droped simultaneously.**

What happens when user issues DELETE command ?

Suppose you have an `emp` table with 100 rows and a multi-user environment involving User 1 and User 2. Both the users are accessing `emp` table



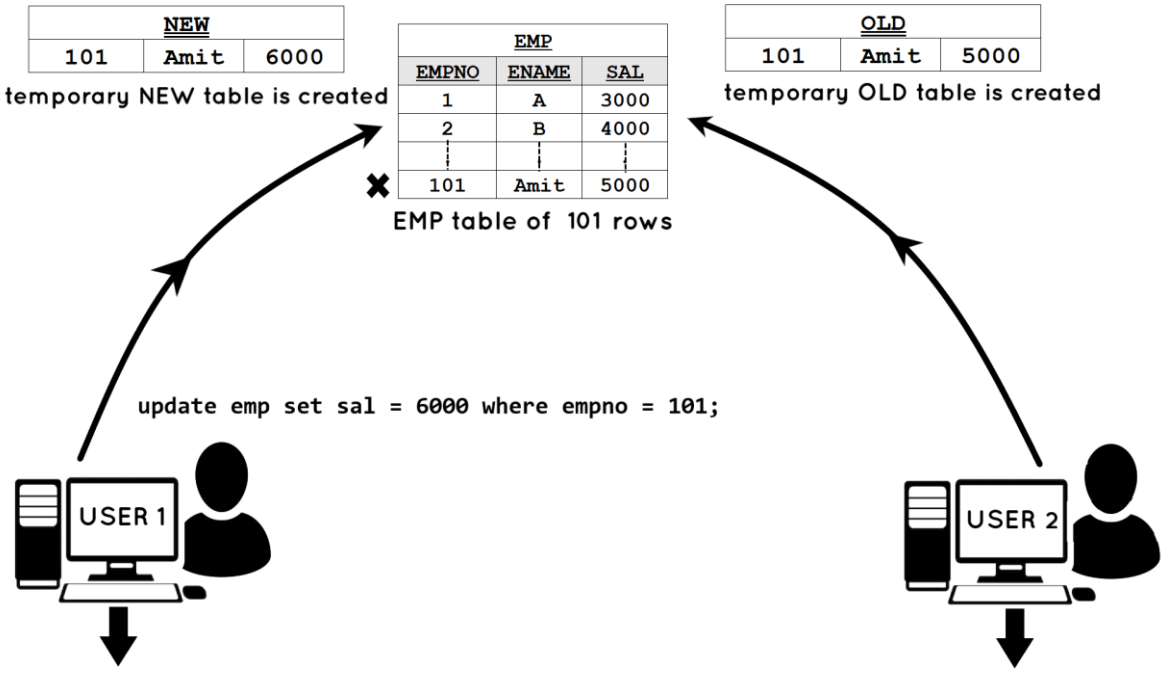
`select * from emp;` • USER 1 can see 101 rows in the emp table	`select * from emp;` • USER 2 can see 101 rows in the emp table
`delete from emp where empno = 101;`	
`select * from emp;` • USER 1 can see 100 rows in the emp table, because row is deleted from emp table	`select * from emp;` • USER 2 can see 101 rows in the emp table, deleted row is still visible for USER 2
`commit;`	
	`select * from emp;` • USER 2 can now see 100 rows in the emp table as deleted row by USER 1 is now deleted for USER 2 when USER 1 gives COMMIT
When multiple users are accessing or modifying the `emp` table simultaneously, database systems use locking mechanisms to ensure data consistency. For instance, if user 1 is updating a row, user 2 might be blocked from accessing that row until the update is complete.	

- When USER 1 issues a delete statement corresponding row will get deleted from EMP table but at the same time MySQL creates a new temporary old table and it stores deleted row in that temporary table
- Therefore, when USER 1 issues select * from emp command he would see 100 rows in the EMP table and at same time when USER 2 issues select * from emp command he would see 101 rows in the table
- Now when USER 1 when issues a commit command it becomes permanent this old table will be dropped from the database and changes made by USER 1 would be visible for USER 2 i.e., user 2 would see 100 rows in the table

What happens when user issues UPDATE command ?

- When you update or delete a row that row is automatically locked for other users i.e. that row (which you updated or deleted) automatically becomes read-only for other users
- Row locking in MySQL is automatic
- Other users can SELECT from that table, they will see the old data before your changes.
- Other users can insert, update or delete other rows in that table, but no other users can update or delete your locked rows till you have issued rollback or commit commands
- The locks are automatically released when you rollback or commit. The entire process is automatic

Suppose you have an `emp` table with 101 rows and a multi-user environment involving User 1 and User 2. Both the users are accessing `emp` table



`select * from emp;` • USER 1 can see 101 rows in the emp table	`select * from emp;` • USER 2 can see 101 rows in the emp table
`update emp set sal = 6000 where empno = 101;`	
`select * from emp;` • USER 1 would see new data of 101 rows (100 rows of emp table plus one row which is NEW table)	`select * from emp;` • USER 2 would see 101 rows (100 rows of emp table plus one row which is OLD table)
`commit;`	
	`select * from emp;` • USER 2 can now see 101 rows in the emp table

- When USER 1 issues an update statement corresponding row will get updated in EMP table but at the same time MySQL creates a two tables NEW table and OLD table.
- Old data is stored in OLD table and updated new data is stored in NEW table
- Updated row is removed from EMP table and EMP table is left with 100 rows
- When USER 1 issues `select * from emp` command he would see 101 rows (100 rows of EMP table plus one updated row which is in NEW table) and at same time when USER 2 issues `select * from emp` command he would see 101 rows (100 rows of EMP table plus one old row which is in OLD table)
- If USER 1 gives commit, new row is added into EMP table and OLD and NEW table are dropped
- If USER 1 gives rollback, old row will come into EMP table and OLD and NEW table are dropped
- USER 2 will see updated rows by USER 1 only when user 1 issues a commit

SQL Plus **USER 1**

```
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 Training       CDAC

SQL> update dept set dname = 'Sleeping'
  2  where deptno = 50;

1 row updated.

SQL> select * from dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 Sleeping       CDAC
```

SQL Plus **USER 2**

```
SQL> select * from dept;

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

SQL> select * from dept;

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

SQL>
```

If USER 2 tries to delete from dept where deptno = 50;, This row is locked in server by USER1 as USER1 has not given commit or rollback command after update statement is executed. So, in the server delete statement is going to wait till the deptno = 50 is available (unlocked) by USER1. deptno = 50 row is automatically released or unlocked when USER1 has given commit or rollback command. As soon as USER1 gives commit, statement delete from dept where deptno = 50; is executed and 1 row is deleted as given below. In this locking is at row level

SQL Plus

```
SQL> select * from dept;

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

SQL> commit;

Commit complete.

SQL>
```

SQL Plus

```
SQL> select * from dept;

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

SQL> delete from dept where deptno = 50;

1 row deleted.

SQL> commit;

Commit complete.

SQL>
```

Optimistic Locking: - automatic row locking mechanism of MySQL

Pessimistic Locking: - manually lock the rows in advance BEFORE issuing UPDATE or DELETE

To lock the rows manually: -

- You will have to use SELECT statement with a FOR UPDATE clause
  e.g.,
  to lock EMP table: -

```
mysql> select * from emp for update;
```

- In the database(backend) all the rows of the emp table are manually locked for other users, whole emp table becomes read-only for other users
- The rows are automatically released when you rollback or commit

NOTE: Other users can INSERT into the table but cannot UPDATE or DELETE till USER 1 have given rollback or commit

Please lock only those rows that you plan to update, hence for this give a suitable WHERE clause

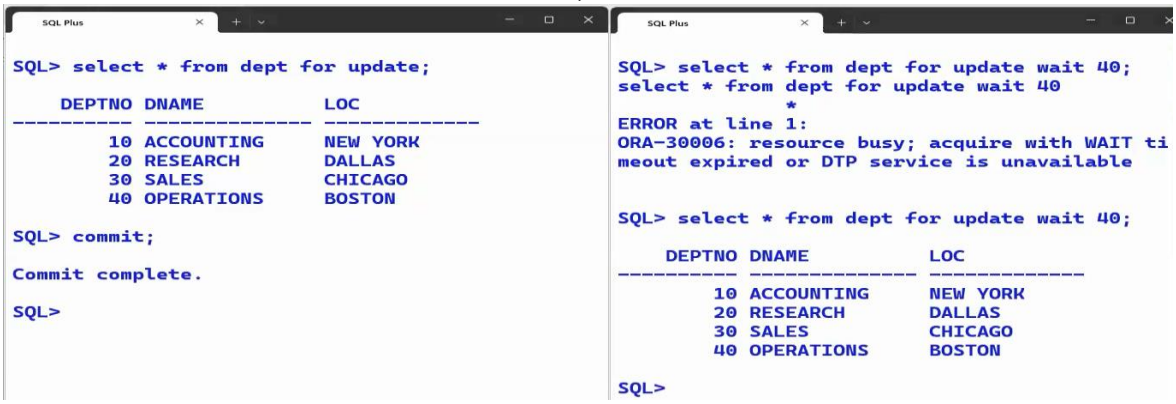
```
mysql> select * from
       where deptno = 10
       for update;
```

```
mysql> select * from
       where deptno = 10
       for update wait;
```

- This will lock the row manually; if row is not available, then it will wait in the Request Queue

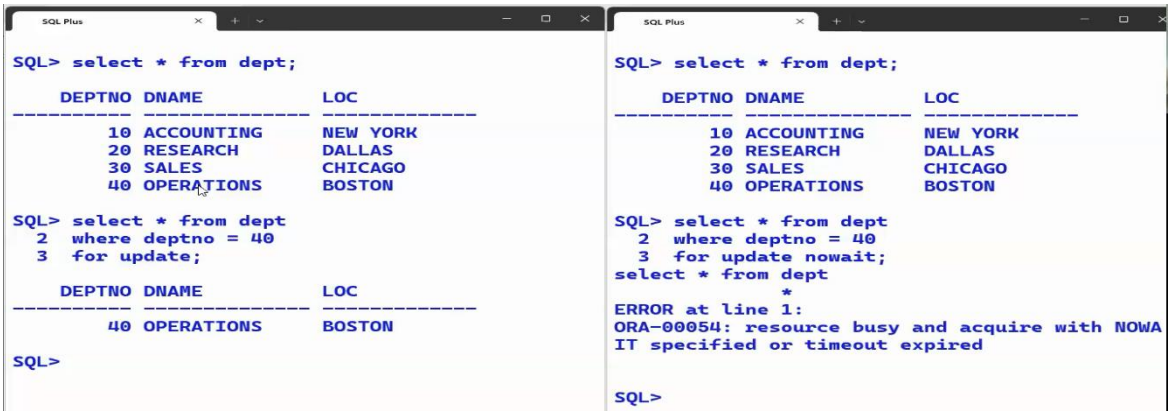
```
mysql> select * from
       where deptno = 10
       for update wait 60;
```

This will lock the row manually; if the row is available lock it; if the row is not available, then it will wait in the Request Queue for a time period of 60 seconds, within 60 seconds if the row becomes available it will lock it, else abort.



```
mysql> select * from
       where deptno = 10
       for update nowait;
```

- This will lock the row manually; if row is not available, then it will not wait in the Request Queue, it will give an error message and abort the operation and you can try again later



- wait/ nowait options not available in MySQL
- wait/ nowait options are available in Oracle

SQL Plus -> is the client s/w for Oracle RDBMS

Run SQL Command Line

```
Username: system
Password: manager
```

In older version of Oracle: -

SQL> connect

```
Username: system
Password: manager
```

When you install Oracle, 3 users are automatically created: -

1. scott
2. system
3. sys

1. scott/tiger
    - demo user
    - regular user
    - connect, resource, and create view privileges
2. system
    - DBA privileges
3. sys
    - Most important user
    - Owner of database
    - Startup database, shutdown database, perform recovery

```
SQL> select * from all_users;
SQL> select username from all_users;
SQL>create user <username> identified by <password>;
SQL>create user pawan identified by student;
SQL>grant connect, resource, create view to pawan;
SQL>disconnect;
SQL>connect
Username: pawan
Password: student
SQL>select * from tab;
Tab → is a system table (stores table names)
SQL>desc emp;
SQL>exit;
```

Day 5

MySQL – SQL- Functions

Character Functions

```
CREATE TABLE emp (
fname VARCHAR(50),
lname VARCHAR(50)
);
INSERT INTO emp (fname, lname)
VALUES
('Arun', 'Purun'),
('Tarun', 'Arun'),
('Sirun', 'Kirun'),
('Nutan', 'Purun');
```

EMP table

FNAME	LNAME
Arun	Purun
Tarun	Arun
Sirun	Kirun
Nutan	Purun

CONCAT() → concatenate (to join) → It allows you to combine two or more strings into a single string

```
select concat(fname, lname) from emp;
```

OUTPUT

```
ArunPurun
TarunArun
SirunKirun
NutanPurun
```

```
select concat(concat(fname, ' '),lname) from emp;
```

- max upto 255 levels for functions within function
  (This limit of SQL can be exceeded with the help of Views)

Uses: Reporting Purpose

OUTPUT

```
Arun Purun
Tarun Arun
Sirun Kirun
Nutan Purun
```

UPPER():
converts a string/text to upper-case.

```
select upper(fname) from emp;
```

OUTPUT

```
ARUN
TARUN
SIRUN
NUTAN
```

Uses: Reporting Purpose

LOWER():
converts a string/text to lower-case

```
select lower(fname) from emp;
```

OUTPUT

```
arun
tarun
sirun
nutan
```

Uses: Reporting Purpose

Solution for Case-insensitive query in Oracle: -

```
select * from emp where lower(fname) = 'arun';
```

INITCAP():

Initial capital, it will convert first letter into upper case and rest of letters into lower case

```
select initcap(fname) from emp;
```

Uses: For Reporting/presentation Purposes

- initcap function is not supported by MySQL
- initcap function is supported by Oracle

OUTPUT

```
Arun
Tarun
Sirun
Nutan
```

EMP table

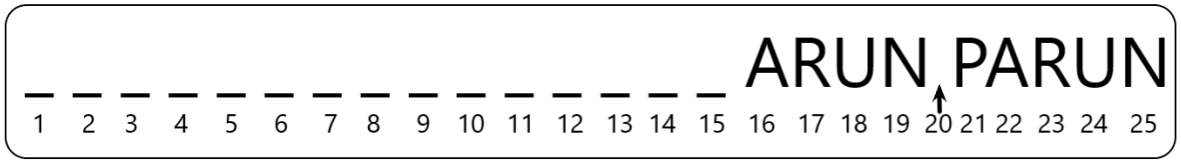
ENAME
Arun Purun
Tarun Arun
Sirun Kirun
Nutan Purun

```
CREATE TABLE emp (
    ename VARCHAR(100)
);
```

```
INSERT INTO emp (ename)
VALUES
('Arun Purun'),
('Tarun Arun'),
('Sirun Kirun'),
('Nutan Purun');
```

LPAD: **Right justification puts blank spaces at the left-hand side**

```
select lpad(ename,25,' ') from emp;
select lpad(ename,25,'*') from emp;
```



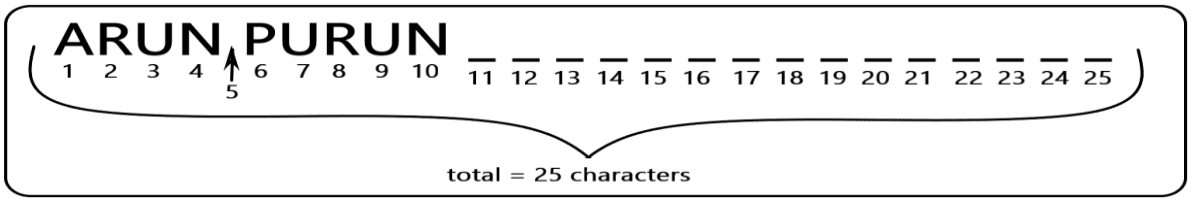
Uses: -

Note: - Blank spaces are counted

- Right Justification
- Reporting purpose e.g., Header, Footer, etc.
- Account/ ATM card/ Credit card number masking
- Cheque Printing

RPAD: Left justification puts blank spaces at the right-hand side

```
select rpad(ename,25,' ') from emp;
select rpad(ename,25,'*') from emp;
```



Uses: -

- Left Justification for numeric data
- To convert varchar to char (convert variable length to fixed length)
- Account/ ATM card/ Credit card number masking
- Cheque Printing
- Center- justification, e.g., lpad(……rpad(……)……)

LTRIM: Removes blank spaces from Left-hand side

```
select ltrim(ename) from emp;
```

Uses: -

- Left Justification e.g., rpad(……..ltrim(…….)……)

RTRIM: Removes blank spaces from Right-hand side

```
select rtrim(ename) from emp;
```

Uses: -

- Convert char to varchar (convert to fixed-length to variable length)
- Right Justification of char data e.g., lpad(……..rtrim(…….)……)

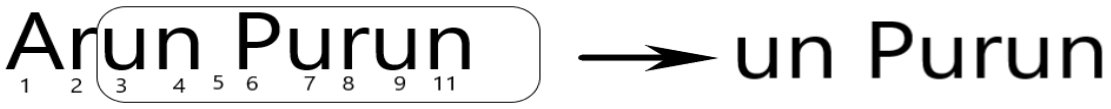
TRIM: Remove blank spaces from both the sides in MySQL

```
select trim(ename) from emp;
```

SUBSTRING: Extracts a part of the string

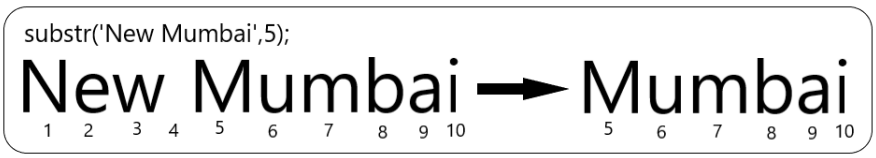
```
select substr(ename, 3) from emp;
```

- Here 3 is Starting position→Starting from and including third letter it will extract a string



OUTPUT

```
un Purun
run Arun
run Kirun
tan Purun
```



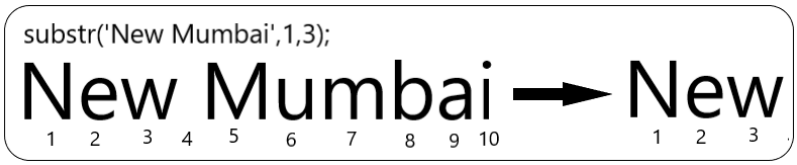

```
select substr(ename, 3, 2) from emp;
```

Here,

- 3 → Starting position
- 2 → no of characters to extract (here it gets 3rd and 4th letter)

OUTPUT

```
un
ru
ru
ta
```



```
select substr(ename, -3) from emp;
```

- Here it will start from right-side/Last end
- -3 is starting position, and get last three letters in every name

OUTPUT

```
run
run
run
run
```


```
select substr(ename,-3,2) from emp;
```

- Here, -3 is starting position, it will start from right-side

2 is number of characters to extract (here it gets -3rd and -2nd letter)

OUTPUT

```
ru
ru
ru
ru
```

REPLACE: Replaces one string with second string

```
select replace(ename,'un','xy') from emp;
```

- Wherever 'un' is there it will replace with 'xy'

NOTE: number of characters in both the string need not to be same
For e.g.,

```
select replace(ename,'un','xyz') from emp;
```

```
select replace(ename,'un','x') from emp;
```

USES:-

- Encoding and Decoding
- Encryption and Decryption
- Masking of ATM
- Card Number

OUTPUT

```
Arxy Purxy
Tarxy Arxy
Sirxy Kirxy
Nutan Purxy
```

INSTR: It stands for Instring, it returns starting position of string

```
select instr(ename,'un') from emp;
```

- If string is not found, then it returns 0

USES: -

- Used to check if one string exists in another string

EMP table
ENAME
Arun
Banerjee
Charlie

```
CREATE TABLE emp (
ename VARCHAR(100)
);
```

```
INSERT INTO emp(ename)
VALUES
('Arun'),
('Banerjee'),
('Charlie');
```

Length: returns the length of string

```
select length(ename) from emp;
```

OUTPUT

```
4
9
7
```

ASCII: returns the ascii value of 1st letter

```
select ascii(ename) from emp;
```

OUTPUT

```
65
66
67
```

```
select ascii(substr(ename,2)) from emp;
```

```
select ascii('z') from emp;
```

OUTPUT

```
122
122
122
```

```
select distinct ascii('z') from emp;
```

OUTPUT

```
122
```

- distinct keyword will suppress duplicates

```
use mysql;
select distinct ascii('z') from user;
```

OUTPUT

```
122
```

```
select ascii('z') from dual;
```

- Dual is a system table (it is automatically created when you install MySQL),
- Dual is a dummy table
- It contains only 1 row and 1 column
- Present in all RDBMS

```
select substr('New Mumbai',5) from dual;
select 'Welcome to CDAC' from dual;
select 3*12 from dual;
```

CHAR: It returns the character corresponding to ascii value

In MySQL: -

```
select char(65 using utf8) from dual;
```
→ A

where utf8 is the given character set for US English else default is binary character set

SOUNDEX:

```
select * from emp where soundex(ename)=soundex('Aroon');
```

Number Functions

EMP table
SAL
1234.567
1852.019
1375.617
1749.156

```
CREATE TABLE emp (
sal FLOAT
);
INSERT INTO emp (sal) VALUES (1234.567);
INSERT INTO emp (sal) VALUES (1852.019);
INSERT INTO emp (sal) VALUES (1375.617);
INSERT INTO emp (sal) VALUES (1749.156);
```

ROUND(): is used to round a number to a specified number of decimal places. If no specified number of decimal places is provided for round-off, it rounds off the number to the nearest integer.

Syntax

```
ROUND(X, D)
```

- **X:** The number which to is rounded.
- **D:** Number of decimal places up to which the given number is to be rounded. It is optional. If not given it round off the number to the closest integer. If it is negative, then the number is rounded to the left side of the decimal point.
- Returns: It returns the number after rounding to the specified places.

```
select round(sal) from emp;
```

OUTPUT

```
1235
1852
1376
1749
```

Rounding off the sal after decimal place

```
select round(sal,1) from emp;
```

OUTPUT

```
1234.6
1852
1375.6
1749.2
```

rounding off the sal, 1 digit after decimal

```
select round(sal,2) from emp;
```

OUTPUT

```
1234.57
1852.02
1375.62
1749.16
```

rounding off the sal, 2 digits after decimal

```
select round(sal,-2) from emp;
```

OUTPUT

```
1200
1900
1400
1700
```

round off 2 digits left hand side of decimal

Rounding off a number when D is not specified.
Rounding a Negative number.

```
SELECT ROUND(-10.11) AS Rounded_Number;
```

```
+----------------+
| Rounded_Number |
+----------------+
|            -10 |
+----------------+
```

In Oracle

`sal number(7,3)`
7 → total number of digits
3 →reserved for decimal
1234.567

EMP table
SAL
1234.567
1852.019
1375.617
1749.156

TRUNCATE():- (removes the decimal point numbers), truncates a number to the specified number of decimal places.

```
select truncate(sal,0) from emp;
OUTPUT
1234
1852
1375
1749
```

```
select truncate(sal,1) from emp;
OUTPUT
1234.5
1852
1375.6
1749.1
```

Shows 1 digit after decimal point

```
select truncate(sal,2) from emp;
OUTPUT
1234.56
1862.01
1375.61
1749.15
```

Shows 2 digits after decimal point

```
select truncate(sal,-2) from emp;
OUTPUT
1200
1800
1300
1700
```

Truncate 2 digits, left-hand side of decimal point

Uses: - Age calculation, Date calculation, Time calculation

NOTE

TRUNCATE, it simply cuts off the decimal part, while ROUND handles rounding to the nearest whole number based on the decimal part's value.

CEIL(): adds 1 to last number by removing decimal point

- short form for ceiling,
- adds 1 to the last number by removing decimal point
- if there is any value in decimal, then it will add 1 to the whole number

```
select ceil(sal) from emp;
OUTPUT
1235
1853
1376
1750
```

Uses: - Bill payments, Interest payments, EMI payment

FLOOR(): removes the decimal and it returns the lower number

```
select floor(sal) from emp;
OUTPUT
1234
1852
1375
1749
```

```
select truncate(3.6,0), floor(3.6), truncate(-3.6,0), floor(-3.6) from dual;
               3          3           -3            -4
```

For Positive number: -
In truncate removes decimal, floor also removes decimal and returns a lower number so there is no difference for positive number.
But
For Negative number: -
if we truncate -3.6 it will remove decimal and give 3 but if we floor -3.6 removes decimal and returns a lower number 4 (because lower number for -3.6 is -4)

SIGN(): returns sign of number

- If number is **negative**, it returns **-1**
- If number is **positive**, it returns **1**
- If number is **0**, it returns **0**

```
select sign(-15) from dual;
-1
```

USES:

- Check if number is +ve or -ve
- Sign(SP-CP)
- Sign(temperature)
- Sign(Blood_group)
- Sign(account_balance)
- Sign(credit_card_bill)
- Sign(medical_report)
- Sign(acceleration)
- Sign(altitude)
- Sign(gravity)
- Sign(stock_market_sensex)
- To find out the greater of two numbers

MOD(): it will return remainder

`select mod(9,5) from dual;`	→ it returns remainder 4
`select mod(8.22,2.2) from dual;`	→ it returns remainder 1.62

SQRT(): it returns square root for positive numbers only

`select sqrt(81) from dual;`	→ 9

POWER(): is used to find the value of a number raised to the power of another number.

It Returns the value of X raised to the power of Y.

Syntax

```
POWER(X, Y)
```

Parameter : This method accepts two parameter which are described below :
X : It specifies the base number.
Y : It specifies the exponent number.

`select power(10,3) from dual;`	-----→ 1000
`select power(10,1/3) from dual;`	-----→ $10^{1/3}$---→returns cube root

ABS(): it always returns a positive numbers

```
select abs(-10) from dual;      → 10
```

x --→ radians
sin(x)
cos(x)
tan(x)

ln(y)---------$\log_e(y)$
ln(n,m)------$\log_n(m)$

Date and Time Functions and Formats (MySQL)

EMP table
HIREDATE
2019-10-15
2019-12-31
2020-01-15

```
CREATE TABLE emp (
hiredate DATE
);
```

```
INSERT INTO emp (hiredate) VALUES
('2019-10-15'),
('2019-12-31'),
('2020-01-15');
```

- 'YYYY-MM-DD' or 'YY-MM-DD'
- 1970 is the year of the Epoch
- 1st Jan 1000 AD to 31st Dec 9999 AD
- Internally date is stored as a fixed-length and it occupies 7 Bytes of storage
- date1-date2

sysdate:

```
select sysdate() from dual;
2023-11-06 19:32:56
```

- returns system date and time
- returns DB Server date and time
- returns date and time when statement executed
- used for clock displays, date and time displays

now

```
select now() from dual;
2023-11-06 19:36:11
```

- returns date and time when statement began to execute
- used to maintain logs of insertion, updations, deletions, programs and other operations, server logs, etc.

```
select sysdate(), now() from dual;
2023-11-06 19:37:49 | 2023-11-06 19:37:49
```

```
select sysdate(), now() from dual;
2023-11-06 19:39:15 | 2023-11-06 19:39:15 |
2023-11-06 19:39:25 | 2023-11-06 19:39:15 |
```

adddate :
this add 24 hrs

```
select adddate(sysdate(),1) from dual;
```
- show date of tomorrow

```
select adddate(sysdate(),2) from dual;
```
- shows date after 2 days

```
select adddate(sysdate(),7) from dual;
```
- shows date of next week

```
select adddate(sysdate(),-1) from dual;
```
- shows date of yesterday

datediff
```
select datediff(sysdate(),hiredate) from emp;
```
- returns the number of days between the two as per the calendar

date_add :
used for month calculation

```
select date_add(hiredate, interval 2 month) from dual;
```
- adds 2 months to the date

```
select date_add(hiredate, interval -2 month) from dual;
```
- substracts 2 months to the date

```
select date_add(hiredate, interval 1 year) from dual;
```
- adds 1 year to the date

last_day:
```
select last_day(hiredate) from dual;
```
- returns last day of the month
- used for attendance calculation

day_name:
```
select dayname(sysdate()) from dual;
```
- returns day of the week

addtime:
```
select addtime('2010-01-15 11:00:00','1') from dual;
```
- adds 1 second to time

```
select addtime('2010-01-15 11:00:00','1:30:45') from dual;
```
- adds 1:30:45 to time

DAY 6

LIST Function

- Independent of datatype (will work with all data types)

EMP table		
ENAME	**SAL**	**COMM**
A	5000	500
B	6000	null
C	null	700

```
CREATE TABLE emp (
ename VARCHAR(50),
sal FLOAT,
comm FLOAT
);
```

```
INSERT INTO emp (ename, sal,
comm) VALUES
('A', 5000, 500),
('B', 6000, NULL),
('C', NULL, 700);
```

```
select * from emp where comm = null;
```

- Here output of above SELECT statement is nothing because null means nothing, when you are searching for 'comm = null' you are searching comm for nothing, and hence if you are searching for nothing then result will also be nothing
- returns null i.e., output is nothing

If you want to find null values in the column use SELECT statement below

```
select * from emp where comm is null;
```

```
select * from emp where comm != null;
```

- returns not null i.e., output is nothing

Any comparison done with null, returns null

- PESSIMISTIC querying: - searching for null values, MySQL is by default optimistic

IS NULL → (Special Operator)

```
select * from emp where comm is null;
```

```
select * from emp where comm is not null;
```

0(zero) is not null (because '0' is having ascii value 48 and 'null' is having ascii value 0)

*Any **operation** done with null, returns null*

```
select sal+comm from emp;
```

OUTPUT

```
5500        →5000+500 = 5500
Null        →6000+null = null
Null        →null+7000 = null
```

- here sal+comm is computed/pseudo column
- Any operation done with null, returns null

If you want above SELECT statement to work then use IFNULL function

IFNULL

```
select sal + ifnull(comm,0) from emp;
```

OUTPUT

```
5500        →5000+500 = 5500
6000        →6000+0 = 6000
Null        →null+700 = null
```

- if comm is null then
  return 0;
  else
  return comm;
  end if;

```
select ifnull(sal,0) + ifnull(comm,0) from emp;
```

OUTPUT

```
5500        →5000+500 = 5500
6000        →6000+0 = 6000
 700        →0+700 = 700
```

```
ifnull(comm,0)              → If comm is null then return 0
ifnull(comm,100)            → If comm is null return 100
ifnull(city,'Mumbai')       → If city is null return Mumbai
ifnull(orderdate,'2023-04-01')
```

GREATEST Function:

It compares and returns greatest among values

EMP table	
SAL	**DEPTNO**
1000	10
2000	10
3000	20
4000	30
5000	40

```
CREATE TABLE emp (
sal FLOAT,
deptno INT
);
```

```
INSERT INTO emp (sal,
deptno)
VALUES (1000, 10),
(2000, 10),
(3000, 20),
(4000, 30),
(5000, 40);
```

```
select greatest(sal,3000) from emp;
```

- Each and every row of 'sal' column is compared with 3000 and returns larger value

OUTPUT

```
3000        1000 compared with 3000
3000        2000 compared with 3000
3000        3000 compared with 3000
4000        4000 compared with 3000
5000        5000 compared with 3000
```

Uses:

- Used to set a lower limit on some value
  e.g., Bonus = 10 % sal, min Bonus = 300

```
select greatest(sal*0.1,300) from emp;
```

```
greatest(col1,col2,col3,…….,col255)
```

```
greatest(val1,val2,val3,…….,val255)
```

```
greatest(num1,num2,num3)
```

```
greatest('str1','str2','str3','str4')
```

```
greatest('date1','date2','date3'
```

LEAST Function:

It compares and returns smallest among values

```
select least(sal,3000) from emp;
```

- Each and every row of 'sal' column is compared with 3000 and returns smaller value

OUTPUT

```
1000        1000 compared with 3000
2000        2000 compared with 3000
3000        3000 compared with 3000
3000        4000 compared with 3000
3000        5000 compared with 3000
```

Uses:

- Used to set an upper limit on some value
  e.g., Cashback = 10% amt, max Cashback = 300

```
select least(amt*0.1,300) from orders;
```

```
least(col1,col2,col3,…….,col255)
```

```
least(val1,val2,val3,…….,val255)
```

```
least(num1,num2,num3)
```

```
least('str1','str2','str3','str4')
```

```
least('date1','date2','date3')
```

CASE expression

```
select
case
when deptno = 10 then 'Training'
when deptno = 20 then 'Exports'
when deptno = 30 then 'Marketing'
else 'Others'
end "DEPTNAME"
from emp;
```

OUTPUT

```
Training
Training
Exports
Marketing
Others
```

```
select
case
when deptno = 10 then 'Training'
when deptno = 20 then 'Exports'
when deptno = 30 then 'Sales'
else 'others'
end
from emp;
```

We don't want above expression to be column heading, so we are giving alias as "DEPTNAME" after end command.

If you don't specify ELSE and if some other value is present in the table, then it returns a null value

- If you don't specify ELSE and if some other value is present in the table, then it returns a null value

```
select
case
when deptno = 10 then 'Training'
when deptno = 20 then 'Exports'
when deptno = 30 then 'Marketing'
else 'Others'
end "DEPTNAME"
from emp;
```

OUTPUT

```
DEPTNAME
Training
Training
Exports
Marketing
Others
```

```
select
case
when deptno = 10 then 'Ten'
when deptno = 20 then 'Twenty'
when deptno = 30 then 'Thirty'
when deptno = 40 then 'Forty'
end "DEPTNAME"
from emp;
```

OUTPUT

```
DEPTNAME
Ten
Ten
Twenty
Thirty
Forty
```

```
if deptno = 10 then HRA = 40% annual
if deptno = 20 then HRA = 30% annual
if deptno = 30 then HRA = 25% annual
else HRA = 20% annual
```

```
select deptno, ename, sal, sal*12 "Annual",
case
when deptno = 10 then sal*12*0.4
when deptno = 20 then sal*12*0.3
when deptno = 30 then sal*12*0.25
else sal*12*0.2
end "HRA"
from emp;
```

```
if sal > 3000 then REMARK = 'High Income'
if sal < 3000 then REMARK = 'Low Income'
if sal = 3000 then REMARK = 'Middle Income'

select ename, sal,
case
when sign(sal-3000) = 1 then 'High Income'
when sign(sal-3000) = -1 then 'Low Income'
else 'Middle Income'
end "REMARKS"
from emp
order by 2;
```

Environment Functions

```
select user() from dual;
```

- Returns a username whatever you have logged in as.
- Used to maintain logs of insertions

```
insert into emp
values(101,'Amit',5000,now(),user());
```

```
show character set;
```

- Set of characters which is there on keyboard is known as character set, If you want to know which all languages are available inside your MySQL installation you can find out from this command

Group/Aggregate Functions

sum (column_name), avg (column_name), min (column_name), max (column_name), count (column_name), count(*), stddev (column_name)

- Group functions are built-in SQL functions that operate on groups of rows and return one value for the entire group.
- Aggregate functions are often used with the GROUP BY clause of the SELECT statement. The GROUP BY clause splits the result-set into groups of values and the aggregate function can be used to return a single value for each group.

EMP

EMPNO	ENAME	SAL	DEPTNO	JOB	MGR
1	Arun	8000	1	M	4
2	Ali	7000	1	C	1
3	Kiran	3000	1	C	1
4	Jack	9000	2	M	**null**
5	Thomas	8000	2	C	4

```
CREATE TABLE emp (
empno INT,
ename VARCHAR(50),
sal FLOAT,
deptno INT,
job CHAR(1),
mgr INT
);
```

```
select
case
when job='M' then 'MANAGER'
when job='C' then 'CLERK'
end "JOB"
from emp;
```

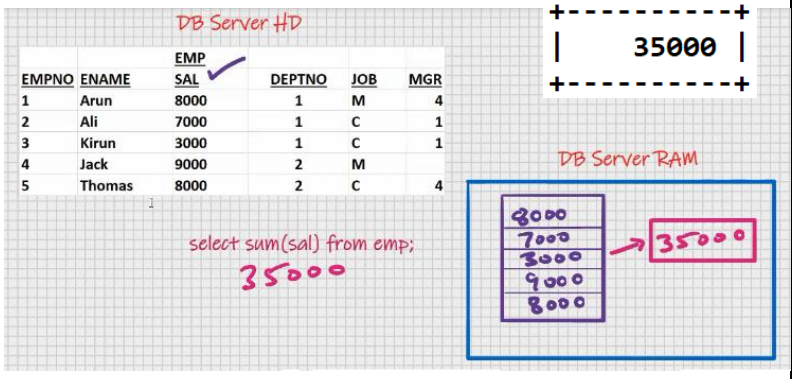
Single Row Function vs Multiple row function

	Single Row Function	Multiple row function
1	It operates on a single row at a time.	It operates on groups of rows.
2	It returns one result per row.	It returns one result for a group of rows.
3	can be used in Select, Where, and Order by	can be used in the select clause only.
4	Math, String and Date functions are examples of single row functions.	Max(), Min(), Avg(), Sum(), Count() and Count(*) are examples of multiple row functions.

SUM(): Returns the sum of the values for the specified column

```
select sum(sal) from emp;
```

```
+----------+
| sum(sal) |
+----------+
|    35000 |
+----------+
```



- These 5 rows of sal column are brought into server RAM. MySQL will put this in 1D array and will go inside for-loop, it will calculate total, and return 35000 (total) will only this much digit will be sent to server to client machine
- Life of array is only for this SELECT statement afterward it does not exist.

EMP table					
EMPNO	**ENAME**	**SAL**	**DEPTNO**	**JOB**	**MGR**
1	Arun	8000	1	M	4
2	Ali	7000	1	C	1
3	Kiran	3000	1	C	1
4	Jack	**null**	2	M	**null**
5	Thomas	8000	2	C	4

```
select sum(sal) from emp;
+----------+
| sum(sal) |
+----------+
|    26000 |
+----------+
```

SUM() Syntax

SELECT SUM(*column_name*)
FROM *table_name*
WHERE *condition*;

```
select sum(ifnull(sal,0)) from emp;
```

OUTPUT

26000

```
+--------------------+
| sum(ifnull(sal,0)) |
+--------------------+
|              26000 |
+--------------------+
```

Null values are not counted by group functions

AVG():Returns the average of the values in the specified column.

```
select avg(sal) from emp;
```

26000/4 → 6500

```
+-----------+
| avg(sal)  |
+-----------+
| 6500.0000 |
+-----------+
```

AVG() Syntax

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

Null values are not counted by group functions

```
select avg(ifnull(sal,0)) from emp;
```

26000/5 → 5200

```
+--------------------+
| avg(ifnull(sal,0)) |
+--------------------+
|          5200.0000 |
+--------------------+
```

MIN(): Returns the smallest value from the specified column

```
select min(sal) from emp;
+----------+
| min(sal) |
+----------+
|     3000 |
+----------+
```

MIN() Syntax

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

```
select min(ifnull(sal,0)) from emp;
+--------------------+
| min(ifnull(sal,0)) |
+--------------------+
|                  0 |
+--------------------+
```

MAX(): Returns the largest value from the specified column

```
select max(sal) from emp;
+----------+
| max(sal) |
+----------+
|     8000 |
+----------+
```

MAX() Syntax

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

```
select max(sal)/min(sal) from emp;
```

→8000/3000 = 2.6

```
+-------------------+
| max(sal)/min(sal) |
+-------------------+
|            2.6667 |
+-------------------+
```

COUNT(): Counts the number of rows in each group.

- The COUNT() function returns the number of rows that matches a specified criterion.
- The COUNT() function will not count the null values

COUNT() Syntax

SELECT COUNT(*column_name*)
FROM *table_name*
WHERE *condition*;

```
select count(sal) from emp;
```

- returns a COUNT of number of rows where SAL is not having a null value

```
+------------+
| count(sal) |
+------------+
|          4 |
+------------+
```

```
select count(*) from emp;
```

- returns a COUNT of total number of rows in the table

```
+----------+
| count(*) |
+----------+
|        5 |
+----------+
```

COUNT(*): This function counts the number of rows in a given column or expression including NULL and Duplicate values.

```
select count(*) - count(sal) from emp;
```

- returns a COUNT of total number of rows not receiving SAL in the table

```
+----------------------+
| count(*) - count(sal) |
+----------------------+
|                    1 |
+----------------------+
```

```
select sum(sal)/count(*) from emp;
→26000/5  →FASTER
```

```
+-------------------+
| sum(sal)/count(*) |
+-------------------+
|         5200.0000 |
+-------------------+
```

```
select avg(ifnull(sal,0)) from emp;
→26000/5  →SLOWER
```

```
+--------------------+
| avg(ifnull(sal,0)) |
+--------------------+
|          5200.0000 |
+--------------------+
```

```
CREATE TABLE emp (
empno INT,
ename VARCHAR(50),
sal FLOAT,
deptno INT,
job CHAR(1),
mgr INT
);
```

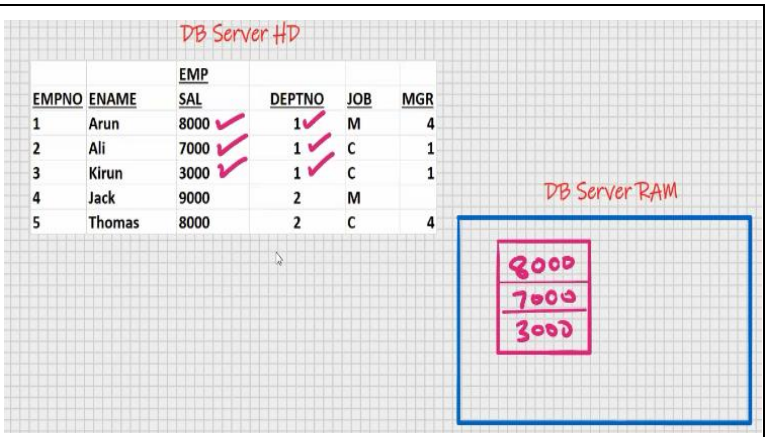
EMP table					
EMPNO	**ENAME**	**SAL**	**DEPTNO**	**JOB**	**MGR**
1	Arun	8000	1	M	4
2	Ali	7000	1	C	1
3	Kiran	3000	1	C	1
4	Jack	9000	2	M	null
5	Thomas	8000	2	C	4

```
INSERT INTO emp (empno, ename, sal, deptno, job, mgr) VALUES
(1, 'Arun', 8000, 1, 'M', 4),
(2, 'Ali', 7000, 1, 'C', 1),
(3, 'Kiran', 3000, 1, 'C', 1),
(4, 'Jack', 9000, 2, 'M', NULL),
(5, 'Thomas', 8000, 2, 'C', 4);
```

```
select sum(sal) from emp
where deptno = 1;
```

OUTPUT

18000




```
select avg(sal) from emp
where job = 'C';
```

OUTPUT

6000

COUNT-QUERY (counting the number of query hits):-

```
select count(*) from emp;
where sal > 7000;
```

OUTPUT

3

- sum(column)
- avg(column)
- min(column) →min(sal), min(ename), min(hiredate)
- max(column) →max(sal), max(ename), max(hiredate)
- count(column) →count(sal), count(ename), count(hiredate)
- count(*)
- stddev(column) ----------standard deviation
- variance(column)

- Aggregate functions can be used in both the SELECT and HAVING clauses (the HAVING clause is covered later in this chapter).
- Aggregate functions cannot be used in a WHERE clause. Its violation will produce the Oracle ORA-00934 group function is not allowed here error message.

You can also use the GROUP BY clause, HAVING clause, and ORDER BY clause to group and sort the results of queries using aggregate functions

Restrictions on the Use of Aggregate Functions

The following restrictions apply to the use of aggregate functions:

- Aggregate functions cannot be nested.
- Aggregate functions can be used only in SELECT clauses or in the HAVING clause of a SELECT statement that includes a GROUP BY clause.

```
select   ename from emp
group by ename
having sum(sal) > 3000;
```

```
+--------+
| ename  |
+--------+
| Arun   |
| Ali    |
| Jack   |
| Thomas |
+--------+
```

- If a SELECT or HAVING clause contains an aggregate function, columns not specified in the aggregate must be specified in the GROUP BY clause. For example:

```
SELECT deptno, max(sal)
FROM emp
GROUP BY deptno;
```

```
+--------+----------+
| deptno | max(sal) |
+--------+----------+
|      1 |     8000 |
|      2 |     9000 |
+--------+----------+
```

The above SELECT statement specifies two columns, deptno and sal, but only sal is referenced by the aggregate function, MAX. The deptno column must be specified in the GROUP BY clause.

mysql> select deptno, max(sal) from emp; ← ERROR

ERROR 1140 (42000): In aggregated query without GROUP BY, expression #1 of SELECT list contains nonaggregated column 'ppisql.emp.DEPTNO'; this is incompatible with sql_mode=only_full_group_by

If a SELECT or HAVING clause contains an aggregate function, columns not specified in the aggregate must be specified in the GROUP BY clause otherwise it gives an error

- You cannot use aggregate functions in a WHERE clause or in a JOIN condition. However, a SELECT statement with aggregate functions in its SELECT list often includes a WHERE clause that restricts the rows to which the aggregate is applied.

SUMMARY REPORTS: -

```
select count(*), min(sal), max(sal), sum(sal), avg(sal) from emp;
```

```
SQL> select count(*), min(sal), max(sal), sum(sal), avg(sal) from emp;

  COUNT(*)   MIN(SAL)   MAX(SAL)   SUM(SAL)   AVG(SAL)
---------- ---------- ---------- ---------- ----------
        14        800       5000      29025 2073.21429
```

Restrictions in group functions

There are three restrictions in group functions

Restriction 1:

- You cannot SELECT column of table along with a Group function

```
select ename, min(sal) from emp;                    <- ERROR
```

```
SQL> select ename, min(sal) from emp;
select ename, min(sal) from emp
       *
ERROR at line 1:
ORA-00937: not a single-group group function
```

```
select count(ename), min(sal) from emp;             <- allowed
```

```
SQL> select count(ename), min(sal) from emp;

COUNT(ENAME)   MIN(SAL)
------------ ----------
          14        800
```

Restriction 2:

- You cannot SELECT a Single-Row function along with a Group function

```
select upper(ename), min(sal) from emp;      <- ERROR
select count(ename), min(sal) from emp;      <- allowed
```

Restriction 3:

- You cannot use Group function in the WHERE clause

```
select * from emp where sal > avg(sal);          <- ERROR
```

(If you want to see all the rows 'sal>avg(sal)' above command won't allow because,

- WHERE clause is used for searching,
- Searching takes place in DB server HD,
- At the time of searching inside the table, 'avg(sal)' does not exist,
- avg(sal) is calculated by MySQL, after the rows are brought inside Server RAM)

If you want to see all the rows 'sal>avg(sal)', you have to write two SELECT statement first find out the 'average sal' then you see 'sal' who is getting greater than 'avg sal'

Group By clause:

Used for grouping

- Group by statement is used to group together any rows of a column with the same value stored in them based on a function specified in the statement
- The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".
- The GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

- WHERE clause → used for searching
- ORDER BY clause → used for sorting
- FOR UPDATE clause → used for locking the rows manually
- GROUP BY clause → used for grouping

MySQL evaluates the GROUP BY clause after the FROM and WHERE clauses but before the HAVING, SELECT, DISTINCT, ORDER BY and LIMIT clauses:


```
select sum(sal) from emp;
```

OUTPUT

```
35000
```

```
select sum(sal) from emp
where deptno = 1;
```

OUTPUT

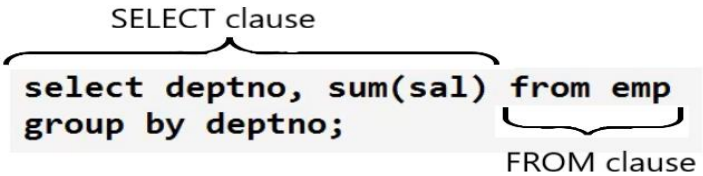
```
18000
```

Finding sum(sal) dept-wise: -

```
select deptno, sum(sal) from emp
group by deptno;
```

OUTPUT

DEPTNO	SUM(SAL)
1	18000
2	17000



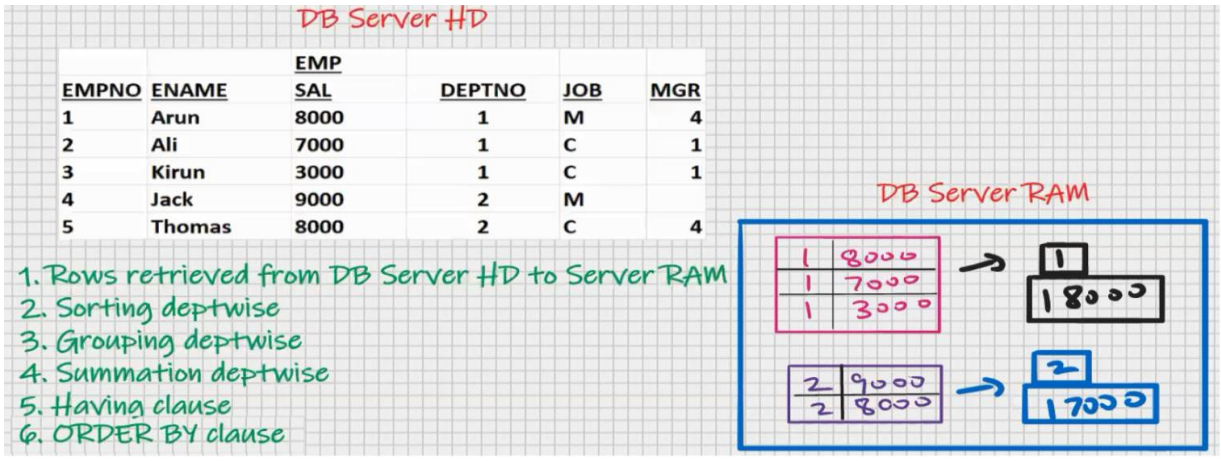
This SELECT statement is executing in 4 steps

1. All SELECT statement execution takes place in Server RAM, so rows of emp table is retrieved from DB Server HD to Server RAM
   (Here only 'SAL' and 'DEPTNO' columns of emp are brought into Server RAM, MySQL will put this into 2D array
   It will go inside FOR loop and calculate respective department totals)

2. Sorting will take place dept-wise internally

3. Grouping is done dept-wise means it will break the array i.e., all the rows of deptno 1 are isolated in one section of RAM, similarly all the rows of deptno 2 are isolated in some section of RAM

4. Summation will take place dept-wise
5. Having clause
6. ORDER BY clause



Rules of Group By clause

```
select deptno, sum(sal) from emp
group by deptno;
```

SELECT clause → `select deptno, sum(sal)`
FROM clause → `from emp`

GROUP BY clause → `group by deptno;`

Rule 1:
Whichever column is present in SELECT clause, it has to be present in the GROUP BY clause

```
select deptno, sum(sal) from emp;   -> ERROR
```

```
SQL> select deptno, sum(sal) from emp;
select deptno, sum(sal) from emp
       *
ERROR at line 1:
ORA-00937: not a single-group group function
```

```
select deptno, sum(sal) from emp
group by deptno;
```

```
SQL> select deptno, sum(sal) from emp
  2  group by deptno;

    DEPTNO   SUM(SAL)
---------- ----------
        10       8750
        30       9400
        20      10875
```

Rule 2:
Besides the group function, whichever column is present in GROUP BY clause, it may or may not be present in SELECT clause

```
select sum(sal) from emp
group by deptno;
```

OUTPUT

```
SUM(SAL)
18000
17000
```

```
select deptno, max(sal) from emp
group by deptno;
```

```
select deptno, min(sal) from emp
group by deptno;
```

```
select deptno, avg(sal) from emp
group by deptno;
```

```
select deptno, count(*) from emp
group by deptno;
```

- Any SELECT statement with a GROUP BY clause is known as a 2D query, because you can plot a graph from the output

```
select job, sum(sal) from emp
group by job;
```

```
select country, sum(sal) from emp
group by country;
```

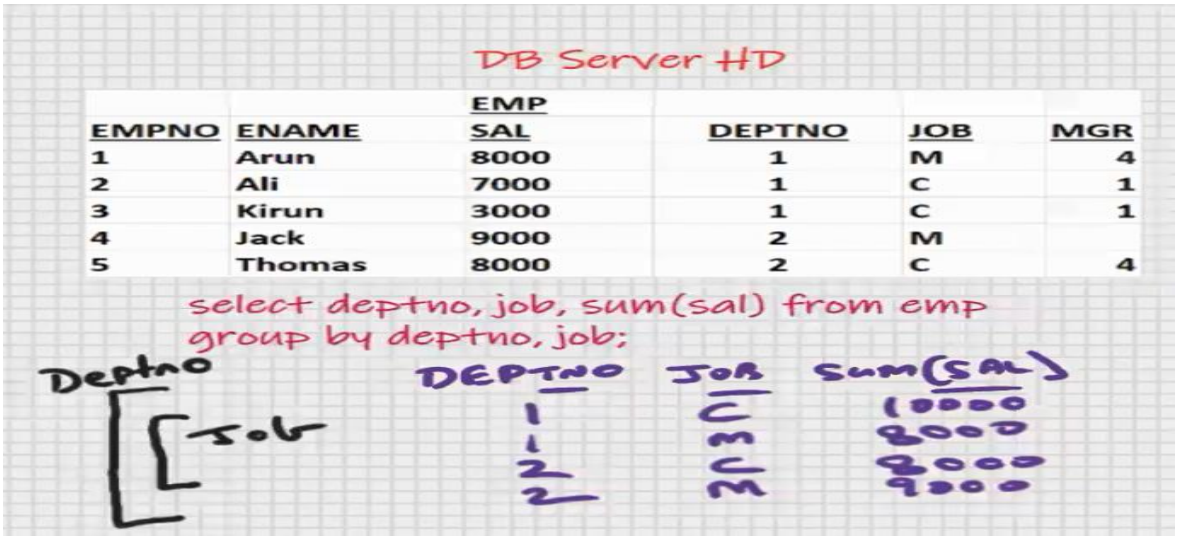
```
select deptno, sum(sal) from emp
where sal > 7000
group by deptno;
```

(SELECT statement execution is from top to bottom, execution is from left to right, WHERE clause is used to retrieve the rows, so only those rows are selected/retrieved whose `'sal>7000'`, only those rows whose `'sal>7000'` are brought in Server RAM then sorting dept-wise, and then grouping dept-wise and finally summation dept-wise takes place)

- WHERE clause is specified BEFORE the GROUP BY clause
- WHERE clause is used for searching,
- Searching takes place in DB server HD
- WHERE clause is used to restrict the rows
- WHERE clause is used to retrieve the rows from DB Server HD to Server RAM

```
select deptno, sum(sal) from emp
where sal > 7000 and job = 'C'
group by deptno;
```

```
select deptno, job, sum(sal) from emp
group by deptno, job;
```



- There is no upper limit on the number of columns in GROUP BY clause

```
select …….
group by country, state, city;
```

- 1 column in GROUP BY clause → 2D query
- 2 columns in GROUP BY clause → 3D query
- 3 columns in GROUP BY clause → 4D query

known as Multi-Dimensional queries
also known as "Spatial" queries

- If you have large number of rows in the table, and if you have large number of columns in GROUP BY clause, then the SELECT statement will be very slow

- The position of columns in SELECT clause, and the position of columns in GROUP BY clause need not be the same
- The position of columns in SELECT clause will determine the position of columns in the output; this you write as per User requirements

```
select deptno, job, sum(sal) from emp
group by deptno, job;

select job, deptno, sum(sal) from emp
group by job, deptno;

select job, deptno, sum(sal) from emp
group by deptno, job;

select sum(sal), job, deptno from emp
group by deptno, job;
```

- The position of columns in GROUP BY clause will determine the sorting order, the grouping order, the summation order, and hence the speed of processing; this you write as per the count of distinct values in the columns

```
select count(distinct deptno) from emp;

select count(distinct job) from emp;

select ……..          <-SLOW
group by state, country, city, district;

select ……..          <-FAST
group by country, state, district, city;
```

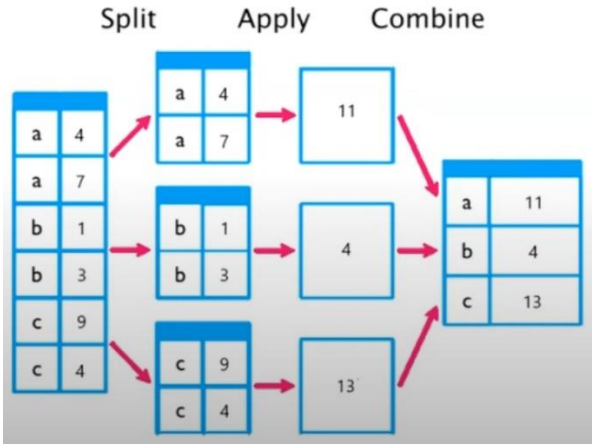
```
select deptno, job, sum(sal) from emp
group by deptno, job;
```

- Whichever column is present in SELECT clause, it has to be present in the GROUP BY clause

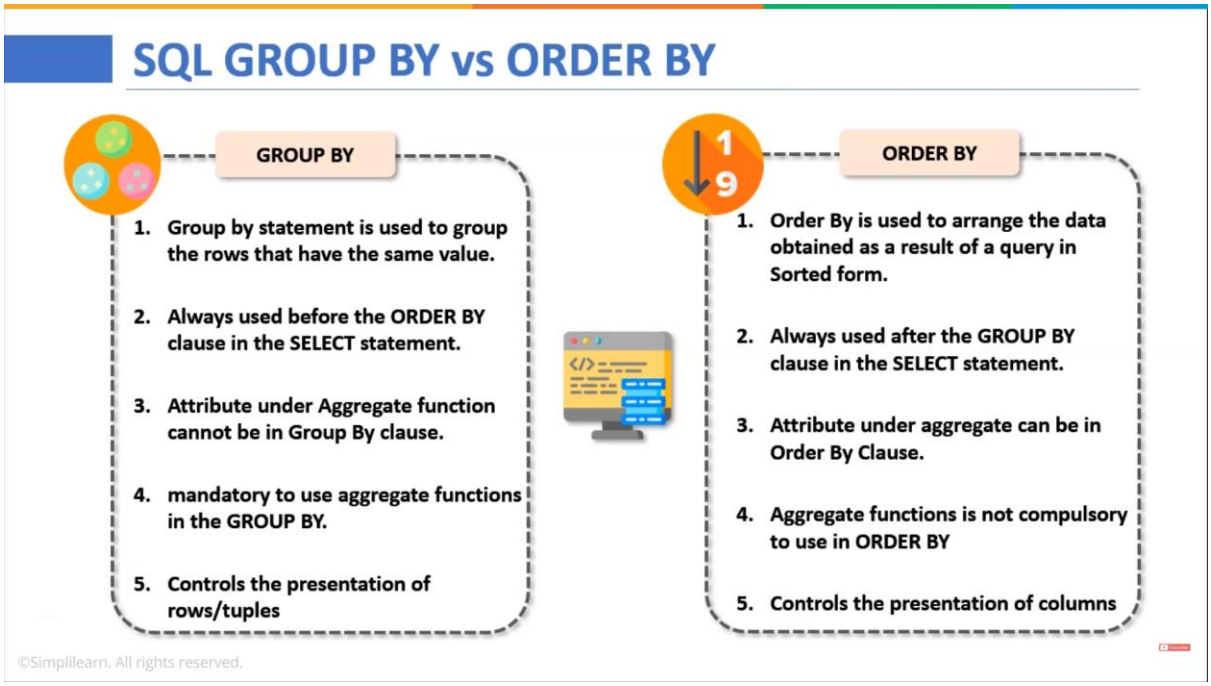
```
select deptno, sum(sal) from emp
group by deptno, job;
```

- Whichever column is present in GROUP BY clause, it may or may not be present in SELECT clause

Group by statement is used to group together any rows of a column with the same value stored in them based on a function specified in the statement



Difference between Group By and Order By



ORDER BY	GROUP BY
Order by keyword sort the result-set either in ascending or descending order.	Group by statement is used to group the rows that have the same value. It is used with aggregate functions for example AVG(), COUNT(), SUM()etc.
It is used to display the information either in ascending order or in descending order	It is used to group the data on the basis of particular column
It is not mandatory to use aggregate functions in order to use Order By command	It is mandatory to use aggregate functions in order to use Group By command
Example: select * from Student order by marks;	Example: select stream, Max(Marks) from Student GROUP BY Stream;

HAVING clause:

- The HAVING clause is used to apply a filter on the result of GROUP BY based on the specified condition.
- The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause
- The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.

```
select deptno, sum(sal) from emp
group by deptno;
```

```
+--------+----------+
| deptno | sum(sal) |
+--------+----------+
|      1 |    18000 |
|      2 |    17000 |
+--------+----------+
```

```
select deptno, sum(sal) from emp
group by deptno
having sum(sal) > 17000;
```

- HAVING clause works after summation is done

```
+--------+----------+
| deptno | sum(sal) |
+--------+----------+
|      1 |    18000 |
+--------+----------+
```

```
select deptno, sum(sal) from emp
where sal > 7000
group by deptno
having sum(sal) > 17000;
```

- WHERE clause is specified BEFORE the GROUP BY clause
- WHERE clause is used for searching
- Searching takes place in DB Server HD
- WHERE clause is used to restrict the rows
- WHERE clause is used to retrieve the rows from DB Server HD to Server RAM
- HAVING clause works after summation is done
- Whichever column is present in SELECT clause, it can be used in HAVING clause

```
select deptno, sum(sal) from emp
group by deptno
having sal > 7000;              ← ERROR
```

- ERROR, because after summation is done 'sal' column does not exist any more what remains in the RAM is deptno and sum(sal) only, instead you should have put sal > 7000 in where clause

```
select deptno, sum(sal) from emp
group by deptno
having deptno = 1;        ← THIS WILL WORK BUT IT WILL BE INEFFICIENT
```

It's recommended that only Group Functions should be used in HAVING clause

```
select deptno, sum(sal) from emp
group by deptno
having sum(sal) > 17000 and sum(sal) < 25000;
```

```
+--------+----------+
| deptno | sum(sal) |
+--------+----------+
|      1 |    18000 |
+--------+----------+
```

```
select deptno, sum(sal) from emp
group by deptno
order by sum(sal);
```

```
+--------+----------+
| deptno | sum(sal) |
+--------+----------+
|      2 |    17000 |
|      1 |    18000 |
+--------+----------+
```

- Shows the ascending order of sum(sal) by writing column name

```
select deptno, sum(sal) from emp
group by deptno
order by 2;
```

efficient way of writing than above select statement

```
+--------+----------+
| deptno | sum(sal) |
+--------+----------+
|      2 |    17000 |
|      1 |    18000 |
+--------+----------+
```

- Shows the ascending order of sum(sal)

1. Rows retrieved from DB Server HD to Server RAM
2. Sorting Dept-wise
3. Grouping Dept-wise
4. Summation Dept-wise
5. Having clause
6. ORDER BY clause

- **ORDER BY clause is the last clause in SELECT statement**

Format of writing SELECT statement

select ..... **from**.....
where.....
group by .....
having.....
order by ..... ;

```
select deptno, sum(sal) from emp
where sal > 7000
group by deptno
having sum(sal) > 15000
order by 1;
```

MATRIX REPORT: -

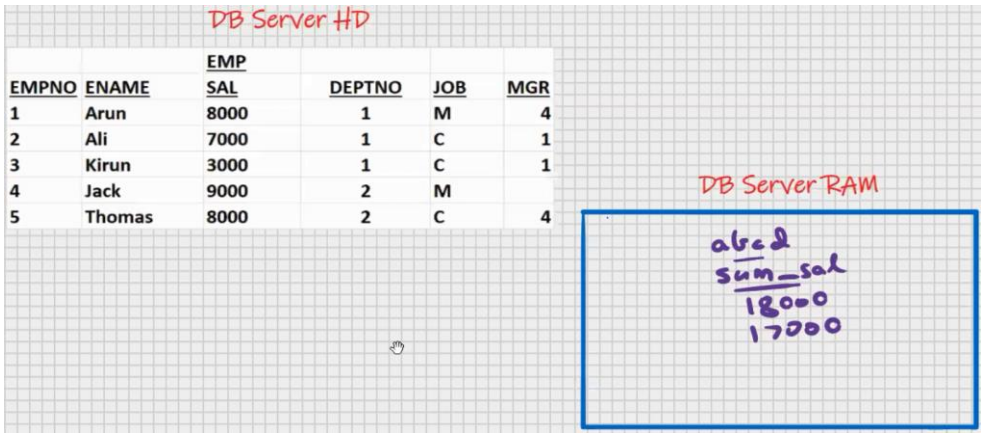
```
select deptno, count(*), min(sal), max(sal), sum(sal) from emp
group by deptno
order by 1;
```

DEPTNO	COUNT(*)	MIN(SAL)	MAX(SAL)	SUM(SAL)
10	3	1300	5000	8750
20	5	800	3000	10875
30	6	950	2850	9400

Difference between WHERE clause and HAVING clause

WHERE clause	**HAVING clause**
The WHERE clause is used to filter records from the table based on the specified condition	The HAVING clause is used to filter records from the groups based on the specified condition.
The WHERE clause implements in row operation	The HAVING clause implements in column operation
The WHERE clause can be used without the GROUP BY clause	The HAVING clause cannot be used without the GROUP By clause
The WHERE clause can be used with SELECT, UPDATE, DELETE statement	The HAVING clause can only be used with the SELECT statement.
The WHERE clause is used to filter rows *before* grouping is performed	The HAVING clause is used to filter rows *after* grouping is performed
The WHERE clause cannot contain aggregate functions	The HAVING clause can contain aggregate functions
GROUP BY clause is used after WHERE clause	GROUP BY clause is used before HAVING clause
Example: select * from student Where MARKS > 40;	Example: Select Stream, MAX(Marks) From students Group By Stream HAVING MAX(Marks)> 40;

In MySQL: -




```
select sum(sal) from emp
group by deptno;
```

```
+----------+
| sum(sal) |
+----------+
|    18000 |
|     8000 |
+----------+
```

```
select max(sum_sal) from
(select sum(sal) sum_sal from emp
group by deptno) abcd;
```

```
+--------------+
| max(sum_sal) |
+--------------+
|        18000 |
+--------------+
```

In Oracle: -

```
select deptno, sum(sal) from emp
group by deptno;
```

OUTPUT

```
deptno  sum(sal)
------  --------
1       18000
2       17000
```

```
select sum(sal) from emp
group by deptno;
```

OUTPUT

```
sum(sal)
--------
18000
17000
```

```
select max(sum(sal)) from emp
group by deptno;
```

- NESTING OF GROUP FUNCTIONS IS SUPPORTED ONLY IN ORACLE RDBMS;
- NOT SUPPORTED BY ANY OTHER RDBMS

```
max(sum(sal))
-------------
18000
```

SQL Order of Execution

The order of execution of an SQL query's clauses is as follows:

1. FROM: First, the table on which the DML operation is performed has to be processed. So, the FROM clause is evaluated first in an SQL Query.

If the query contains **JOIN clauses**, tables are combined by merging rows involved, before FROM clause. So, JOIN predates FROM in statements with JOIN.

Now table data is acquired before any filter or group is done. So subsequent clauses can be evaluated based on this data which is much smaller than the original tables before JOIN operations. This also processes the subqueries.

SQL may create a temporary table internally to evaluate this. To make code memory efficient, it is good to have simpler table JOINs it would be highly memory intensive.

2. WHERE: After the table data on which other operations take place is processed by JOIN and FROM clause, WHERE clause is evaluated. WHERE clause filters the rows based on conditions from the table evaluated by the FROM clause. This WHERE clause discards rows that don't satisfy the conditions, thus reducing the rows of data that need to be processed further in other clauses.

3. GROUP BY: If the query has a GROUP BY clause, it is executed next. Here, the data is grouped based on the common value in the column specified in the GROUP BY clause. This reduces the number of rows further equal to no of distinct values in the GROUP BY column. This helps to calculate aggregate functions.

4. HAVING: If the query had a GROUP BY clause, the HAVING clause is evaluated immediately after GROUP BY. HAVING clause is not compulsory for GROUP BY. Similar to WHERE operations, this clause also filters the table group processed before by the GROUP BY clause. This HAVING also discards rows that don't satisfy the conditions, thus reducing the rows of data that need to be processed further in other clauses

5. SELECT: The SELECT is executed next after GROUP BY and HAVING. It computes expressions (arithmetic, aggregate, etc.) and aliases given in the SELECT clause. The computation is now performed on the smallest dataset after much filtering and grouping operations done by previous clauses.

6. DISTINCT: The DISTINCT clause is executed after evaluating expressions, and alias references in the previous step. It filters any duplicate rows and returns only unique rows.

7. ORDER BY: After executing all the above clauses, the data to be displayed or processed is computed. Now ORDER BY is executed to sort it based on particular column(s) either in ascending or descending order. It is left associative, that is it is sorted based on the first specified column and then by the second, and so on.

8. LIMIT/OFFSET: At last, after the order of data to be processed is evaluated, LIMIT and OFFSET clauses are evaluated to display only the rows that fall within the LIMIT. So, it is generally not recommended to LIMIT only certain rows from many rows evaluated before, since It is not efficient and waste of computation.

Order of Execution of SQL Queries Example

Let's understand the order of Execution of SQL query with an example.

Assume there is a table with the name "orders" which contains data about orders made by customers, like order_ID, customer_ID, customer_city, order_date, and total_amount of the order.

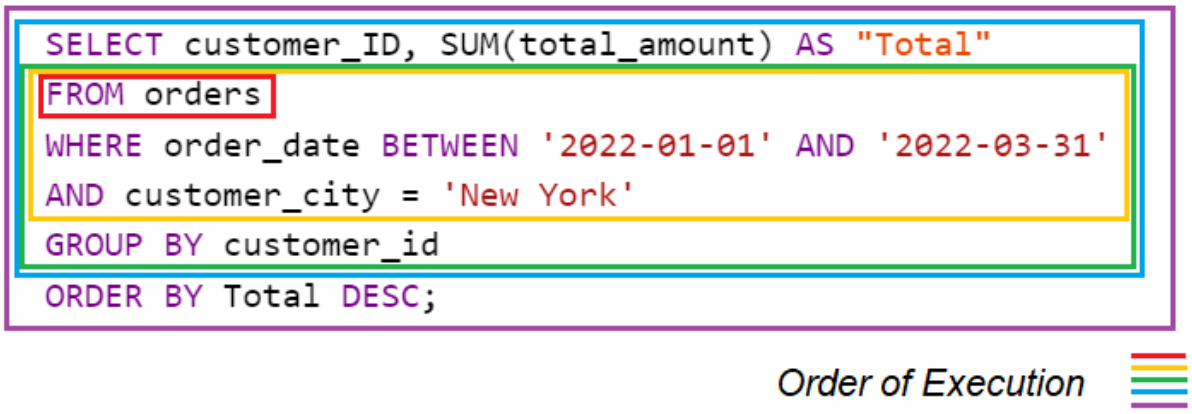
The below query retrieves the total amount of all orders named "TOTAL" made by customers who are living in New York and placed their orders between the dates January 1, 2022, and March 31, 2022, sorted by the total amount in descending order.

Query:

```
SELECT customer_ID,SUM(total_amount) AS "Total"
FROM orders
WHERE order_date BETWEEN '2022-01-01' AND '2022-03-31'
AND customer_city = 'New York'
GROUP BY customer_id
ORDER BY Total DESC;
```

The above query is executed in the following way.

- First, the FROM clause is executed to identify the table involved, which is "orders."
- Next, the WHERE clause is executed to filter the rows based on the conditions, which here is the date range using BETWEEN operation and customer_city value.
- Next, the GROUP BY clause is executed which groups the rows based on Customer_ID. This makes us use the SUM aggregate function now.
- Next, the SELECT clause is executed which retrieves the columns customer_ID and SUM of total_amount corresponding to each ID with the alias name "TOTAL".
- At last, the ORDER BY clause is executed which sorts the results based on "TOTAL" in descending order. The order of evaluation is described in the subset view below.



Order of Execution in SQL

If the above is executed in a different order, some incorrect results would have been evaluated or evaluation would be slow. For instance, if the ORDER BY clause is evaluated before the WHERE clause, the data would be sorted first then grouping would have taken place and It would have returned incorrect results.

By understanding the order of execution of SQL and applying optimization techniques, efficient queries that give the desired results faster and more accurately can be developed.

The order of query execution in SQL is the order in which the various clauses in a SQL statement are executed. Generally, the order is as follows:

SQL processes queries in the order: FROM, JOIN, WHERE, GROUP BY, HAVING, SELECT, DISTINCT, ORDER BY, and finally, LIMIT/OFFSET.

Stages of SQL Order of Execution

1. FROM/JOIN : Getting Data
2. WHERE : Row Filter
3. GROUP : BY Grouping
4. HAVING : Group Filter
5. SELECT : Return Expression
6. DISTINCT: Removes duplicates
7. ORDER BY : Order & Paging
8. LIMIT/OFFSET : Order & Paging

Clause	Function
FROM / JOIN	When you write any query, SQL starts by identifying the tables for the data retrieval and how they are connected.
WHERE	It acts as a filter; it filters the record based on the conditions specified by the users.
GROUP BY	The filtered data is grouped based on the specified condition.
HAVING	It is similar to the WHERE clause but applied after grouping the data.
SELECT	The clause selects the columns to be included in the final result.
DISTINCT	Remove the duplicate rows from the result. Once you apply this clause, you are only left with distinct records.
ORDER BY	It sorts (increasing/decreasing/A->Z/Z->A) the results based on the specified condition.
LIMIT / OFFSET	It determines the number of records to return and from where to start.

In SQL, the execution order of queries is determined by the logical query processing order, which consists of the following steps:

1. FROM clause: The tables are accessed and joined, and the result set is created.
2. WHERE clause: The rows that meet the search condition are filtered.
3. GROUP BY clause: The rows are grouped based on the specified column(s).
4. HAVING clause: The groups that meet the search condition are filtered.
5. SELECT clause: The columns to be displayed are selected.
6. ORDER BY clause: The rows are sorted based on the specified column(s).

The actual order is as follows:

1. **FROM and JOIN**: determine the base data of interest.

2. **WHERE**: filter base data of interest to retain only the data that meets the where clause.

3. **GROUP BY**: group the filtered data by a specific column or multiple columns. Groups are used to calculate aggregates for selected columns.

4. **HAVING**: filter data again by defining constraints on the columns that we grouped by.

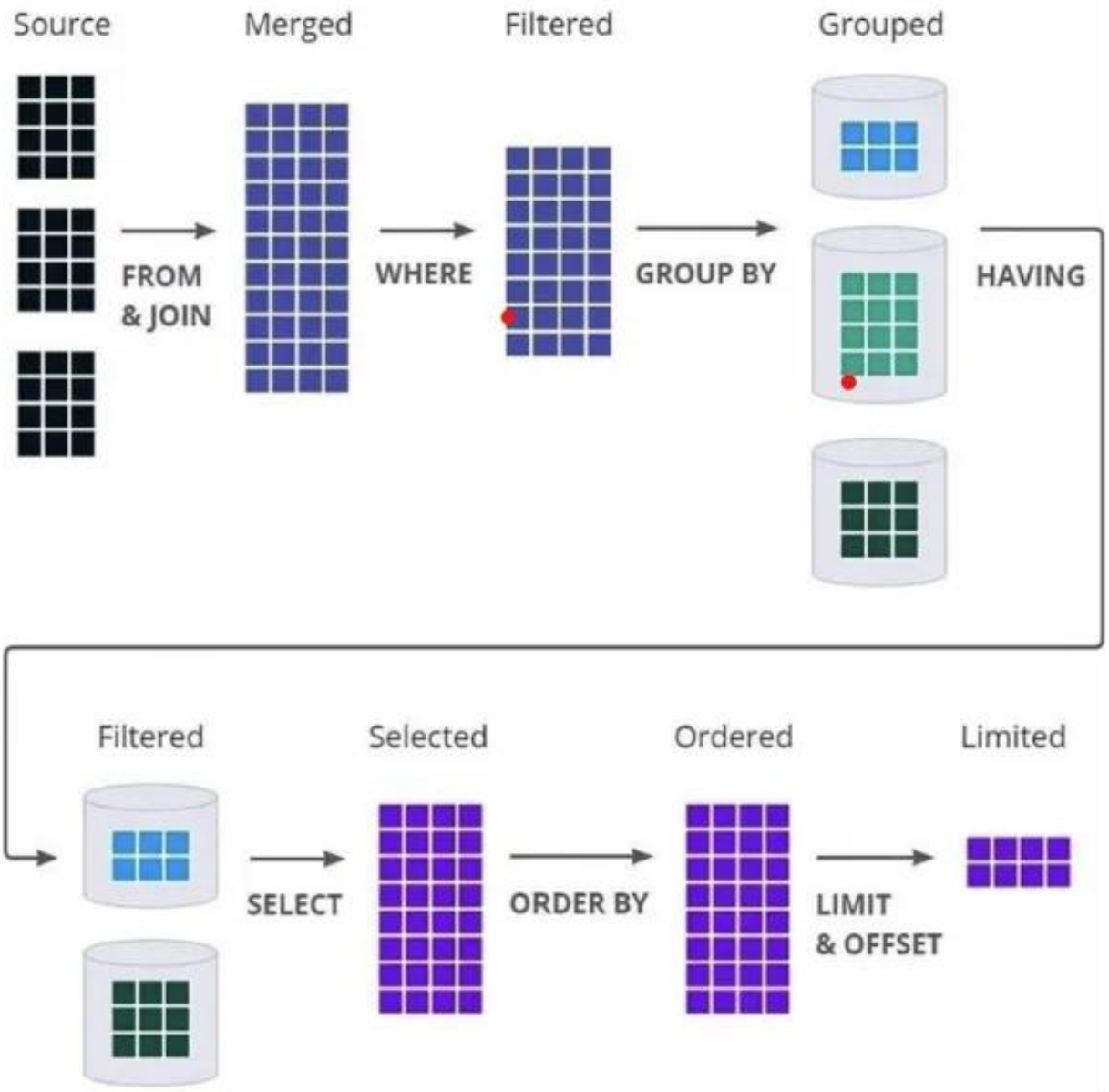
5. **SELECT**: select a subset of columns from the filtered grouped result. This is also where **WINDOW function aggregations** happen.

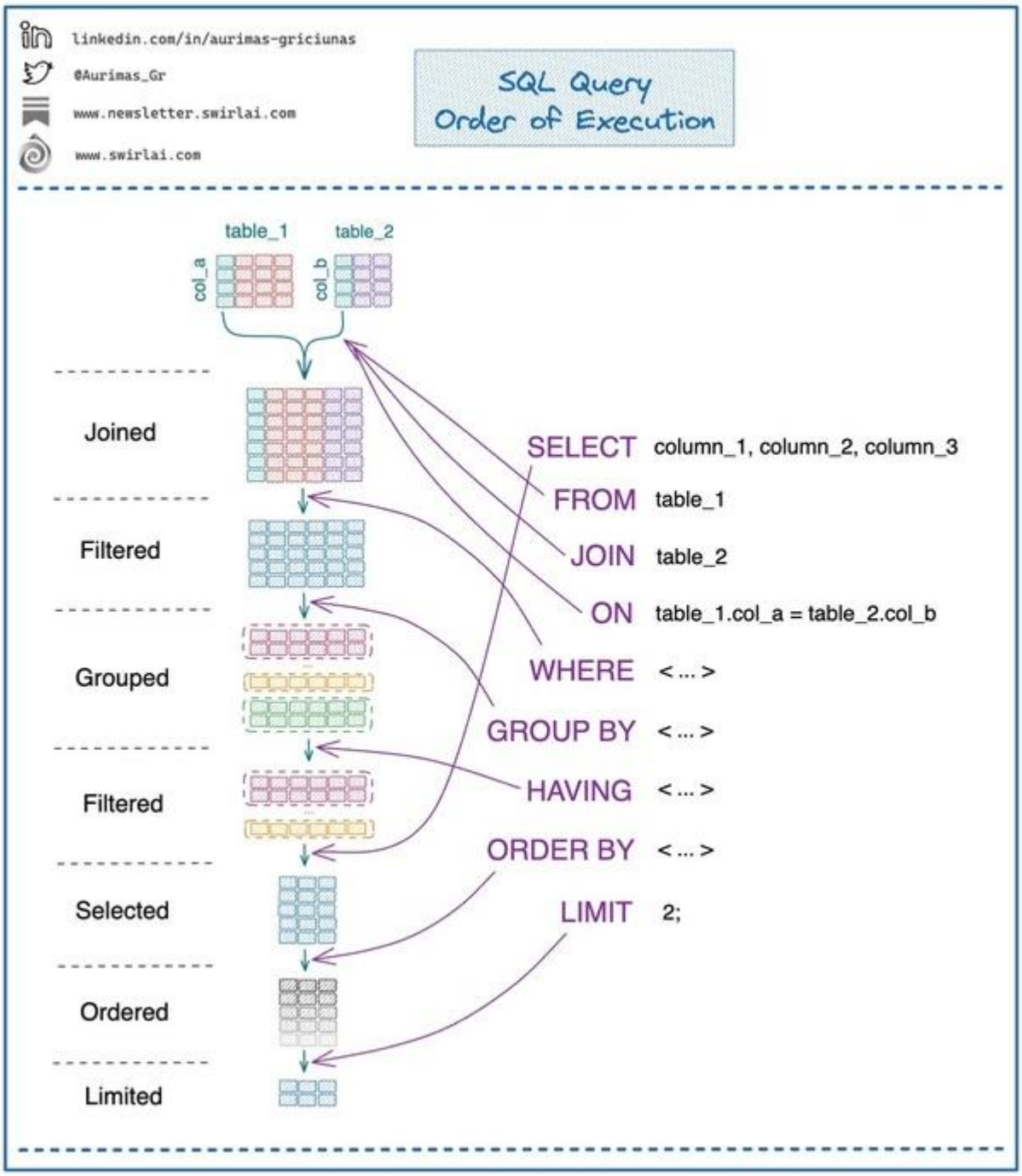
6. **ORDER BY**: order the result by one or multiple columns.

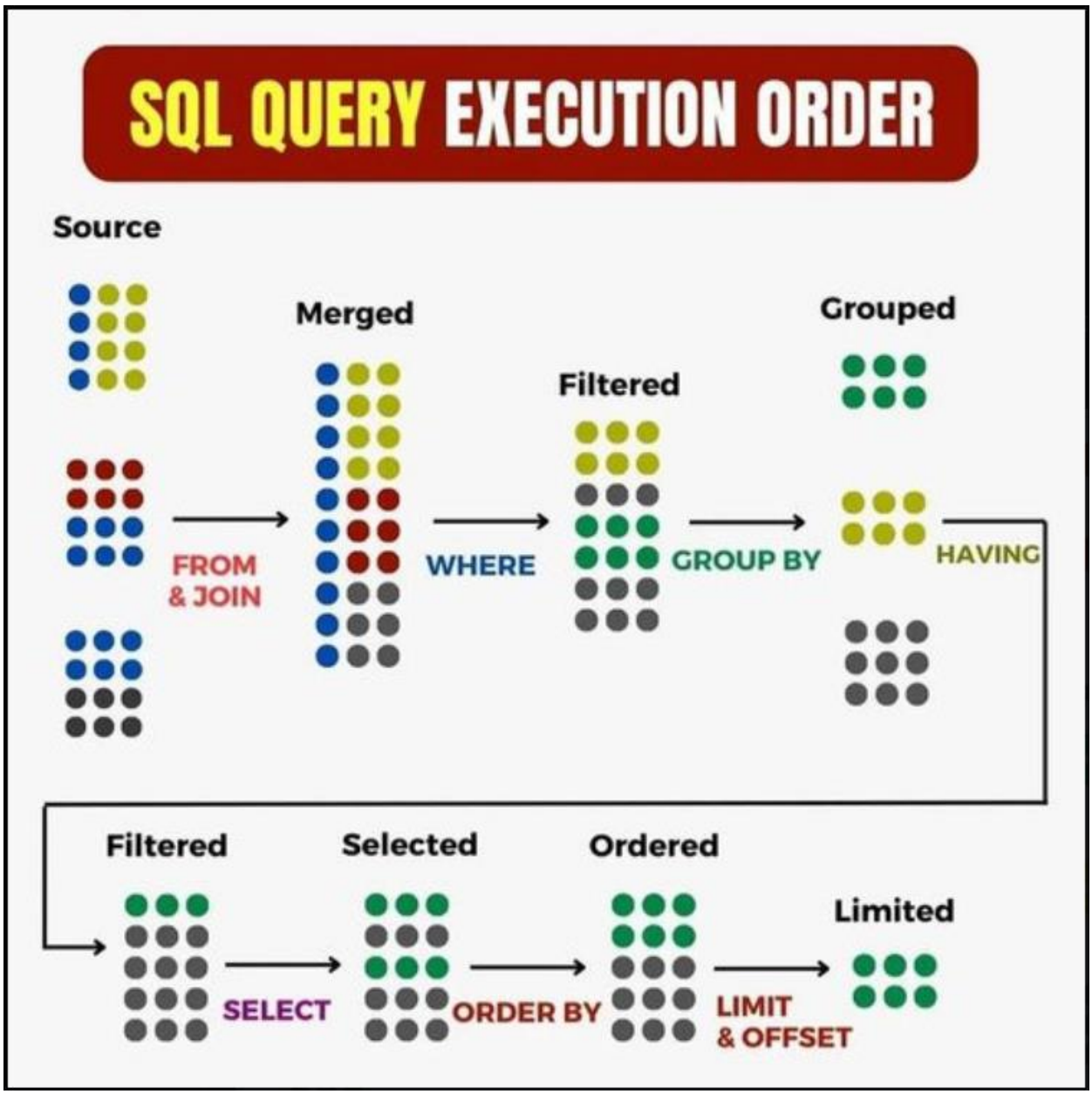
7. **LIMIT**: only retain the top n rows from the ordered result.

Trick to Remember the order of execution of query
"FROM WHERE GROUPS HAVE SELECT ORDERS LIMITED"

SQL Query Execution Order




linkedin.com/in/aurimas-griciunas
@Aurimas_Gr
www.newsletter.swirlai.com
www.swirlai.com
SQL Query
Order of Execution
table_1
table_2
col_a
col_b
Joined
Filtered
Grouped
Filtered
Selected
Ordered
Limited
SELECT column_1, column_2, column_3
FROM table_1
JOIN table_2
ON table_1.col_a = table_2.col_b
WHERE <...>
GROUP BY <...>
HAVING <...>
ORDER BY <...>
LIMIT 2;


SQL QUERY EXECUTION ORDER
Source
Merged
Filtered
Grouped
FROM
& JOIN
WHERE
GROUP BY
HAVING
Filtered
Selected
Ordered
Limited
SELECT
ORDER BY
LIMIT
& OFFSET

Customers Table

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

Orders Table

order_id	item	amount	customer_id
1	Keyboard	400	4
2	Mouse	300	4
3	Monitor	12000	3
4	Keyboard	400	1
5	Mousepad	250	2

Problem Statement: Find the amount spent by each customer belonging to the USA.

```
SELECT Customers.first_name, Customers.last_name, SUM(Orders.Amount)
as Amount
FROM Customers
JOIN Orders ON Customers.customer_id = Orders.customer_id
WHERE Customers.country = 'USA'
GROUP BY Customers.first_name, Customers.last_name
ORDER BY Amount DESC;
```

Output

first_name	last_name	Amount
John	Doe	400
Robert	Luna	250

Explanation

- **FROM** and **JOIN:** We start by identifying the '**Customers**' and '**Orders**' tables and joining them on '**customer_id**'.
- **WHERE:** It will filter the record to include only those where '**country**' = '**USA**'.
- **GROUP BY:** Group the remaining entries (after filtering by WHERE clause) by '**first_name**' and '**last_name**'.
- **SELECT:** SELECT the '**first_name**', '**last_name**', and the sum of '**Amount**' for each group.
- **ORDER BY:** Finally, the result is sorted by '**Amount**' in descending order.

Now, let's take an example and reshuffle the order of execution in sql.

Case-1: Let you want to filter the record based on the 'Amount' using the WHERE clause.

```
SELECT Customers.first_name, Customers.last_name, SUM(Orders.Amount)
as Amount
FROM Customers
JOIN Orders ON Customers.customer_id = Orders.customer_id
WHERE Orders.Amount >300
GROUP BY Customers.first_name, Customers.last_name
ORDER BY Amount DESC;
```

Output

first_name	last_name	Amount
David	Robinson	12000
John	Doe	400
John	Reinhardt	400

Case-2: Filter the record based on the 'Amount' using the HAVING clause.

```
SELECT Customers.first_name, Customers.last_name, SUM(Orders.Amount)
as Amount
FROM Customers
JOIN Orders ON Customers.customer_id = Orders.customer_id
GROUP BY Customers.first_name, Customers.last_name
HAVING Amount > 300
ORDER BY Amount DESC;
```

Output

first_name	last_name	Amount
David	Robinson	12000
John	Reinhardt	700
John	Doe	400

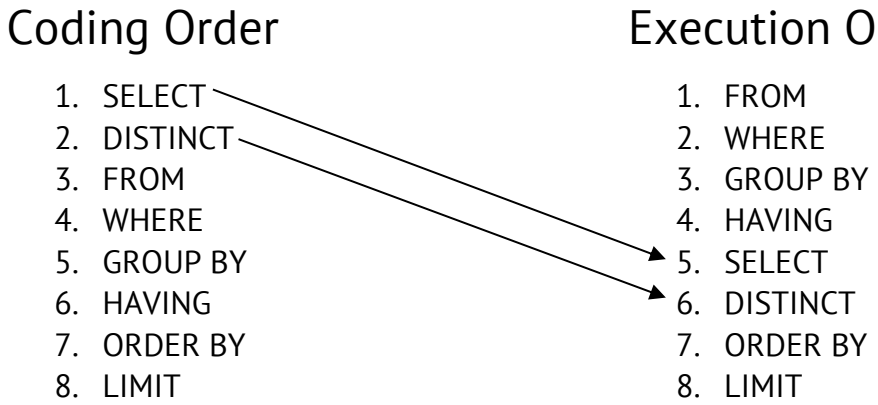
Now, let's see what happened in both cases:

Since the WHERE clause is processed before the SELECT clause in the Order of Execution. So, in the first case, SQL won't recognize the Amount and will give the error.
It just filters out the record of the customer who purchased orders greater than 300.

However, the best way to filter the aggregate function is to use the HAVING clause. Since the HAVING clause is processed after the GROUP BY clause. So, in the second case, the HAVING clause filters the group to include only those where the total Amount is greater than 300.

Tips for Writing Efficient SQL Queries

- The first thing you must know while writing the SQL queries is the correct order of SQL query execution.
    - Since a lot of people think SQL processes queries from top to bottom as they have written.
    - But SQL processes queries in the order: FROM, JOIN, WHERE, GROUP BY, HAVING, SELECT, DISTINCT, ORDER BY, and finally, LIMIT/OFFSET.
- One of the common mistakes is using aliases defined in the SELECT clause within the WHERE clause.
    - Because SQL processes the WHERE clause before the SELECT clause.
- Use the HAVING clause if you need to filter your query based on the result of an aggregate function.
- While joining multiple tables, start with the smallest table or the table that allows you to filter out the most data early on.



EMP table

EMPNO	ENAME	SAL	DEPTNO	JOB	MGR
1	Arun	8000	1	M	4
2	Ali	7000	1	C	1
3	Kirun	3000	1	C	1
4	Jack	9000	2	M	null
5	Thomas	8000	2	C	4

```
CREATE TABLE emp (
empno INT,
ename VARCHAR(50),
sal FLOAT,
deptno INT,
job CHAR(1),
mgr INT
);
INSERT INTO emp (empno, ename, sal, deptno, job, mgr)
VALUES
(1, 'Arun', 8000, 1, 'M', 4),
(2, 'Ali', 7000, 1, 'C', 1),
(3, 'Kiran', 3000, 1, 'C', 1),
(4, 'Jack', 9000, 2, 'M', NULL),
(5, 'Thomas', 8000, 2, 'C', 4);
```

DEPT table

DEPTNO	DNAME	LOC
1	TRN	Bby
2	EXP	Dlh
3	MKTG	Cal

```
create table dept(
DEPTNO int,
DNAME varchar(20),
LOC varchar(20)
);

insert into
dept(DEPTNO,DNAME,LOC)
values
(1,'TRN','Bby'),
(2,'EXP','Dlh'),
(3,'MKTG','Cal');
```

DAY 7

MYSQL – SQL – JOINS (V. imp)

EMP table					
EMPNO	ENAME	SAL	DEPTNO	JOB	MGR
1	Arun	8000	1	M	4
2	Ali	7000	1	C	1
3	Kirun	3000	1	C	1
4	Jack	9000	2	M	null
5	Thomas	8000	2	C	4

DEPT table		
DEPTNO	DNAME	LOC
1	TRN	Bby
2	EXP	Dlh
3	MKTG	Cal

DEPTNO 3 is having null employee

Data Redundancy: Unnecessary duplication of data and that is wastage of HD space

JOINS: -

All the data is not stored in one table; the data is stored in multiple tables; if you want to view/combine the columns of 2 or more tables, then you will have to write a join

```
select ename, deptno from emp;
```

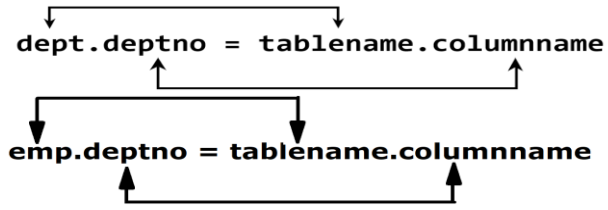
OUTPUT

```
+--------+--------+
| ename  | deptno |
+--------+--------+
| Arun   |      1 |
| Ali    |      1 |
| Kirun  |      1 |
| Jack   |      2 |
| Thomas |      2 |
+--------+--------+
```

```
select dname, ename from emp, dept
where dept.deptno = emp.deptno;
```

…….(CONDITION)

SYNTAX



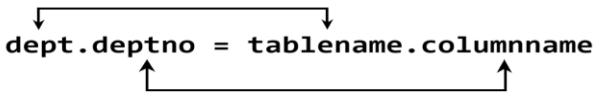
OUTPUT

DNAME	ENAME
TRN	Arun
TRN	Ali
TRP	Kirun
EXP	Jack
EXP	Thomas

dept → driving table
emp → driven table

The driving table is the last table in the FROM clause moving from right to left or first NESTED SELECT.

```
select dname, ename from emp, dept
where dept.deptno = emp.deptno;
```



OUTPUT

DNAME	ENAME
TRN	Arun
TRN	Ali
TRP	Kirun
EXP	Jack
EXP	Thomas

All SELECT statement execution takes place from top to bottom and from left to right, last table i.e., 'dept' is driving table and second last table i.e., 'emp table' is driven table. Here,
dept → driving table
emp → driven table

- Order does not matter here i.e., you can write column name and respective table name randomly in SELECT statement, MySQL is intelligent software, internally it will describe two tables and it will understand that dname is to be returned from dept table and ename is to be returned from emp table
- MySQL will fetch the first row from driving table which is dept table in which deptno = 1 and dname TRN is there, then it will go to emp table (driven) table and search for deptno = 1 in the entire table (searching the whole table is called as full table scan). In searching, 3 rows are present of deptno = 1,
  here deptno = 1 in emp table and deptno = 1 in dept table are same and hence where condition is satisfied and it will return dname i.e., 'TRN' with corresponding ename values 'Arun', 'Ali', and 'Kirun' in the output
- Now MySQL will fetch the second row from driving table which is dept table in which deptno = 2 and dname EXP is there, then it will go to emp table (driven) table and search for deptno = 2 in the entire table (searching the whole table is called as full table scan). In searching 2 rows are present of deptno = 2,
  here deptno = 2 in emp table and deptno = 2 in dept table are same and hence where condition is satisfied and it will return dname i.e., 'EXP' with corresponding ename values 'Jack' and 'Thomas', like this it will continue and comeback to dept table
- Again, it will fetch dept no = 3 and dname MKTG is there in dept table which is driving table and then it will go to emp table and will search entire for dept no = 3 (full table scan). But dept no = 3 is not present in entire table, and hence where condition is not satisfied and nothing will be returned
- Last at the end of dept table will stop since all rows of deptno of dept table ends

INTERNALLY WHOLE EXECUTION IS NESTED FOR LOOP

To make output more presentable: -

```
select dname, ename from emp, dept
where dept.deptno = emp.deptno
order by 1;
```

- output will come in ascending order of dname

```
+-------+--------+
| dname | ename  |
+-------+--------+
| EXP   | Jack   |
| EXP   | Thomas |
| TRN   | Arun   |
| TRN   | Ali    |
| TRN   | Kirun  |
+-------+--------+
```

The position of columns in SELECT clause will determine the position of columns in output: -

```
select dname, ename from emp, dept
where dept.deptno = emp.deptno;
```

DNAME	ENAME
TRN	Arun
TRN	Ali
TRP	Kirun
EXP	Jack
EXP	Thomas

```
select ename, dname from emp, dept
where dept.deptno = emp.deptno;
```

ENAME	DNAME
Arun	TRN
Ali	TRN
Kirun	TRP
Jack	EXP
Thomas	EXP

The comparison/position of columns in the WHERE clause is insignificant, the output will be the same:-

```
select dname, ename from emp, dept
where emp.deptno = dept.deptno;
```

OR

```
select dname, ename from emp, dept
where dept.deptno = emp.deptno;
```

DNAME	ENAME
TRN	Arun
TRN	Ali
TRP	Kirun
EXP	Jack
EXP	Thomas

In order for the join to work faster, the driving table should be table with lesser number of rows:

```
select dname, ename from emp, dept      <-FAST
where dept.deptno = emp.deptno;

select dname, ename from dept, emp      <-SLOW
where dept.deptno = emp.deptno;
```

In order for you to write join,

- The common column in both the tables (e.g., deptno), **the column name need not to be the same in both the tables**, because the same column can have a different meaning in some other table; e.g., Export for one country is Import for another country, purchase for one person is sale for another person (expense for you is income for canteen)
- The common column in both the tables (e.g., deptno), **the datatype has to match in both the tables** and there has to be some sensible relation between the tables on whose basis you are making join

```
select dname, ename from emp, dept
where dept.x = emp.y;
```

```
select dname, ename from emp, dept
where dept.deptno = emp.deptno
order by dname;
```

```
select dname, ename from emp, dept
where dept.deptno = emp.deptno
order by 1;
```

```
select dname, loc, ename, job, sal from emp, dept
where dept.deptno = emp.deptno
order by 1;
```

To see all columns from both the tables: -

```
select * from emp, dept
where dept.deptno = emp.deptno
order by 1;
```

```
        (*) (ERROR)
select deptno, dname, loc, ename, job, sal from emp, dept
where dept.deptno = emp.deptno
order by 1;
```

- ERROR: Column ambiguously defined

```
select dept.deptno, dname, loc, ename, job, sal from emp, dept
where dept.deptno = emp.deptno
order by 1;
```

OR

```
select emp.deptno, dname, loc, ename, job, sal from emp, dept
where dept.deptno = emp.deptno
order by 1;
```

It is a good programming practice to use tablename**.columnname** for all the columns, it makes the SELECT statement more readable:-

```
select dept.deptno, dept.dname, dept.loc, emp.ename, emp.job, emp.sal
from emp, dept
where dept.deptno = emp.deptno
order by 1;
```

```
select deptno, sum(sal) from emp
group by deptno;
```

```
+--------+----------+
| deptno | sum(sal) |
+--------+----------+
|      1 |    18000 |
|      2 |    17000 |
+--------+----------+
```

```
select dname, sum(sal) from emp, dept
```

```
+-------+----------+
| dname | sum(sal) |
+-------+----------+
| TRN   |    18000 |
| EXP   |    17000 |
+-------+----------+
```

```
where dept.deptno = emp.deptno
group by dname;
```

```
select upper(dname), sum(sal) from emp, dept
where dept.deptno = emp.deptno
group by upper(dname);
```

```
select dname, sum(sal) from emp, dept
where dept.deptno = emp.deptno
group by dname
having ……
order by ……;
```

5 Types of Joins:-

EQUIJOIN (NATURAL JOIN)

- Join based on equality condition
- show matching rows of both the tables
- mostly frequently used join (more than 90%) and hence it is also known as NATURAL JOIN
- Data is not stored in one table; data is stored in multiple tables; if you want to view/combine the columns of 2 or more tables then you will write EQUIJOIN
- Uses:
  e.g.,
  DNAME, ENAME
  CNAME, SNAME etc.

```
select dname, ename from emp, dept
where dept.deptno = emp.deptno;
```

dept → driving table
emp → driven table

dname	ename
TRN	Arun
TRN	Ali
TRN	Kirun
EXP	Jack
EXP	Thomas

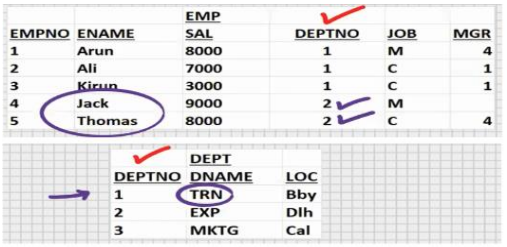
INEQUIJOIN(NON – EQUI JOIN)

- Join based on inequality condition
  (NON EQUI JOIN performs a JOIN using comparison operator other than equal(=) sign like !=,>, <, >=, <= with conditions.)
- show non-matching rows of both the tables
- Used in Exception Reports e.g., who are the employees who don't belong to training

```
select dname, ename from emp, dept
where dept.deptno != emp.deptno;
```

dept → driving table
emp → driven table

first row, deptno=1, TRN →



DNAME	ENAME
TRN	Jack
TRN	Thomas
EXP	Arun
EXP	Ali
EXP	Kirun
MKTG	Arun
MKTG	Ali
MKTG	Kirun
MKTG	Jack
MKTG	Thomas

second row, deptno=2, EXP→

third row, deptno=3, MKTG→


```
select dname, ename from emp, dept
where dept.deptno != emp.deptno and dname = 'TRN';
```

dname	ename
TRN	Jack
TRN	Thomas

```
SELECT dname,ename FROM emp, dept
WHERE emp.deptno = dept.deptno AND dname = 'EXP';
```

dname	ename
EXP	Jack
EXP	Thomas

OUTER JOIN

- Join with (+) sign in oracle
- shows **matching rows of both the tables** plus **non-matching rows of outer table**
- Outer table → table which is on Outer side of (+) sign
- Outer table → table which is on Opposite side of (+) sign
- Uses: -
  Parent-Child Report (Master- Detail Report)

Types of Outerjoin:-

A. Half Outerjoin ((+) sign on any one side, i.e., LHS or RHS)
   a. Right Outerjoin
   b. Left Outerjoin

B. Full Outerjoin ((+) sign on both the sides) (not supported directly)

A. Half Outerjoin

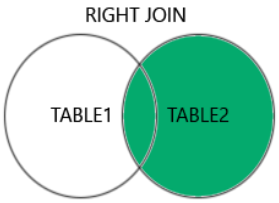
a. Right Outerjoin

Returns all records from the right table, and the matched records from the left table

```
select dname, ename from emp, dept
where dept.deptno = emp.deptno (+);
```

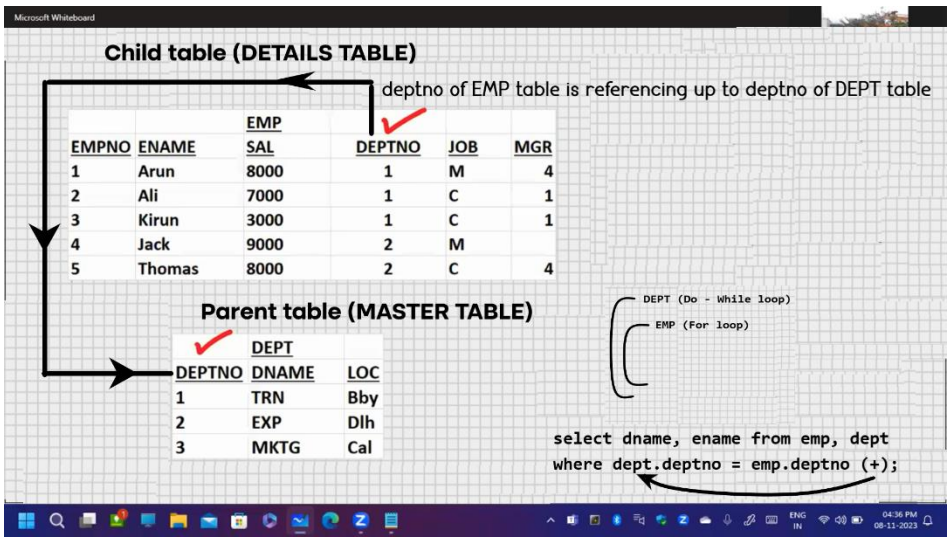
LHS RHS

← Right Outerjoin •dept →



driving table

- emp → driven table
- Child table (DETAILS table) = EMP table
- Parent table (MASTER table) = DEPT table
- EMP table is storing deptno of DEPT table i.e., deptno of EMP table is referencing up deptno of DEPT table;
- DEPT table is not storing deptno of EMP table;
- Because you put (+) sign on RHS of where condition, it will show all the rows of DEPT table even if the where condition is not satisfied



OUTPUT

DNAME	ENAME
TRN	Arun
TRN	Ali
TRN	Kirun
EXP	Jack
EXP	Thomas
MKTG	null

DEPT table goes (outer loop) Do While loop,
EMP table goes (inner loop) For Loop

b. Left Outerjoin

Returns all records from the left table, and the matched records from the right table

EMP table

EMPNO	ENAME	SAL	**DEPTNO**	JOB	MGR
1	Arun	8000	**1**	M	4
2	Ali	7000	**1**	C	1
3	Kirun	3000	**1**	C	1
4	Jack	9000	**2**	M	null
5	Thomas	8000	**2**	C	4
6	Scott	6000	**4**		

DEPT table

DEPTNO	DNAME	LOC
1	TRN	Bby
2	EXP	Dlh
3	MKTG	Cal

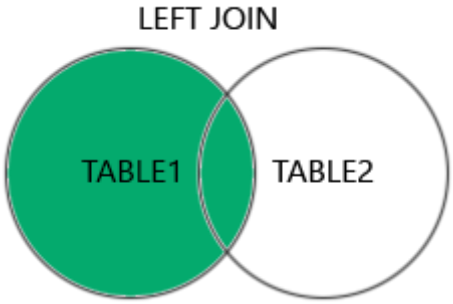
Suppose the table is having 6th row

```
select dname, ename from emp, dept
where dept.deptno (+) = emp.deptno;     ← Left Outerjoin
```

- Because you put (+) sign on LHS of where condition, it will show all the rows of EMP table even if the where condition is not satisfied

NOTE: DEPT table goes (outer loop) For loop,
EMP table goes (inner loop) Do While loop

DNAME	ENAME
TRN	Arun
TRN	Ali
TRN	Kirun
EXP	Jack
EXP	Thomas
Null	Scott



B. Full Outerjoin

- shows matching rows of both the tables
  plus
  non-matching rows of both the tables

```
select dname, ename from emp, dept
where dept.deptno = emp.deptno (+)
union
select dname, ename from emp, dept
where dept.deptno (+) = emp.deptno;
```

DNAME	ENAME
TRN	Arun
TRN	Ali
TRN	Kirun
EXP	Jack
EXP	Thomas
MKTG	Null
null	Scott

(+) sign for Outerjoin is supported ONLY by Oracle RDBMS; not by MySQL and other RDBMS

ANSI syntax

ANSI syntax for Full Outerjoin: -
(Supported by all RDBMS including Oracle, except for MySQL)

```
select dname, ename from emp full outer join dept
on (dept.deptno = emp.deptno);
```

ANSI syntax for Right Outerjoin: -
(Supported by all RDBMS including Mysql and Oracle)

```
select dname, ename from emp right outer join dept
on (dept.deptno = emp.deptno);
```

ANSI syntax for Left Outerjoin: -
(Supported by all RDBMS including Mysql and Oracle)

```
select dname, ename from emp left outer join dept
on (dept.deptno = emp.deptno);
```

To create Full Outerjoin in MySQL: -

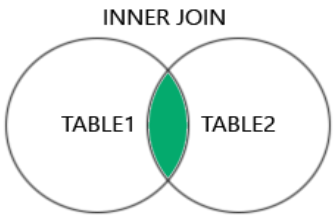
```
select dname, ename from emp right outer join dept
on (dept.deptno = emp.deptno)
union
select dname, ename from emp left outer join dept
on (dept.deptno = emp.deptno);
```

INNER JOIN: Returns records that have matching values in both tables

- DO NOT MENTION IN INTERVIEWS UNLESS EXPLICITLY ASKED BY INTERVIEWER if interviewer asks you then answer the following given below
- **By default, every join is an Inner join**; by using a (+) sign in Oracle, or using the keyword 'Outer' is what makes it an Outer join

```
SELECT * FROM emp INNER JOIN dept
ON
emp.DEPTNO = dept.DEPTNO;
```

EMPNO	ENAME	SAL	DEPTNO	JOB	MGR	DEPTNO	DNAME	LOC
1	Arun	8000	1	M	4	1	TRN	Bby
2	Ali	7000	1	C	1	1	TRN	Bby
3	Kirun	3000	1	C	1	1	TRN	Bby
4	Jack	9000	2	M	NULL	2	EXP	Dlh
5	Thomas	8000	2	C	4	2	EXP	Dlh



EMP table

EMPNO	ENAME	SAL	**DEPTNO**	JOB	MGR
1	Arun	8000	**1**	M	4
2	Ali	7000	**1**	C	1
3	Kirun	3000	**1**	C	1
4	Jack	9000	**2**	M	null
5	Thomas	8000	**2**	C	4

DEPT table

DEPTNO	DNAME	LOC
1	TRN	Bby
2	EXP	Dlh
3	MKTG	Cal

CARTESIAN JOIN (CROSS JOIN): Returns all records from both tables.

- join without a WHERE condition
- every row of driving table is combined with each and every row of driven table (it shows all the Combinations) because there is no WHERE CONDITION
- it will take the Cross product of 2 tables
- **USES:**
  In the STUDENTS table you have all the student's names. In the SUBJECTS table you have all the subjects' names; when you are printing the marksheet for the students, then every student name is combined with each and every subject name, you will require CARTESIAN JOIN

```
select dname, ename from emp, dept;
```

dept → driving table
emp → driven table

- FAST (the I/O between Server HD and Server RAM is less)

```
select dname, ename from dept, emp;
```

emp → driving table
dept → driven table

- SLOW (the I/O between Server HD and Server RAM is more)

DNAME	ENAME
TRN	Arun
TRN	Ali
TRN	Kirun
TRN	Jack
TRN	Thomas
EXP	Arun
EXP	Ali
EXP	Kirun
EXP	Jack
EXP	Thomas
MKTG	Arun
MKTG	Ali
MKTG	Kirun
MKTG	Jack
MKTG	Thomas

- Interchanging driving and driven table output will not change.
- Preferably driving table should be taken with less number of rows if you want to work it faster
- **Cartesian join is the Fastest join, because there is no WHERE clause hence there is no searching involved**

SELF JOIN

- joining a table to itself
- used when parent and child column, both are present in same table
- based on Recursion
- this is the SLOWEST join
- Uses: ENAME, MGRNAME

```
select a.ename, b.ename from emp b, emp a
where a.mgr = b.empno;
```

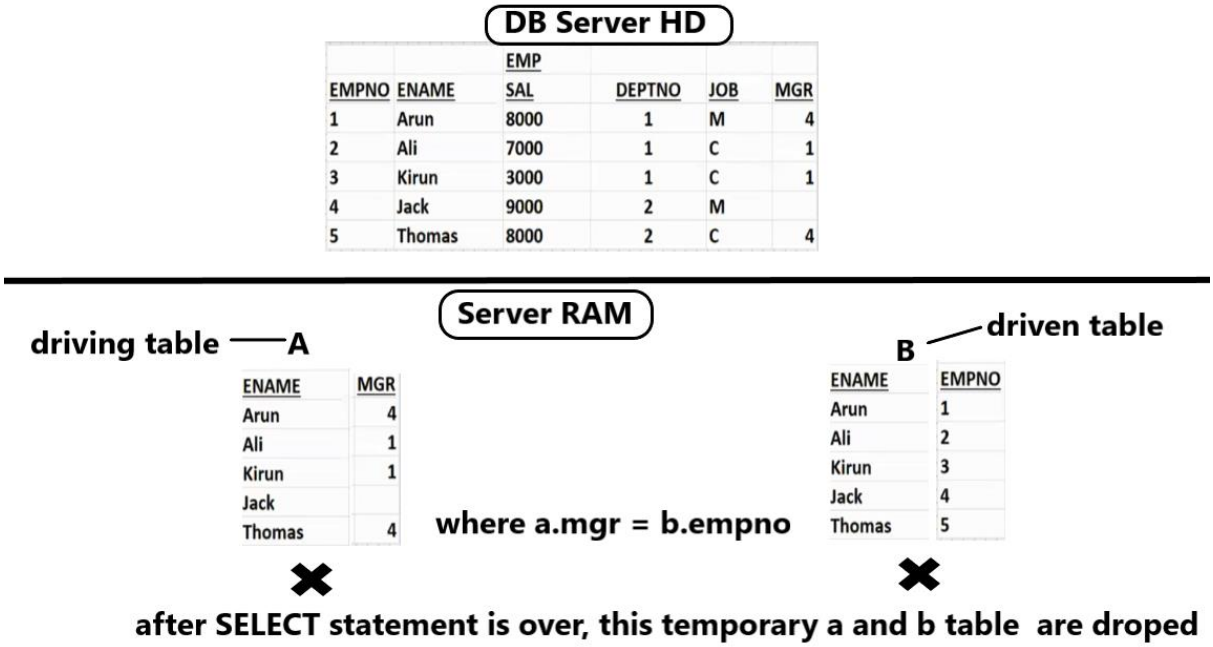
Here a and b are alias for emp table name

EMP

EMPNO	ENAME	SAL	DEPTNO	JOB	MGR
1	Arun	8000	1	M	4
2	Ali	7000	1	C	1
3	Kirun	3000	1	C	1
4	Jack	9000	2	M	
5	Thomas	8000	2	C	4

Parent Column (EMPNO) — Child Column (MGR)

a.ename	b.ename
Arun	Jack
Ali	Arun
Kirun	Arun
Thomas	Jack



Here we have relationship between EMPNO and MGR column, both parent column and child column are present in same table, MGR column is child column (because MGR is storing value derived from EMPNO column, Therefore EMPNO is parent column and MGR is child column

(How to find out Child column and parent column?

Find out who (MGR) is storing whose (EMPNO) value; who is referencing whom; who is dependent on whom), here EMPNO is independent column i.e., parent column

Since EMPNO and MGR are in same table we have to maintain two copies of table in database, we will select EMPNO from one table and MGR from other table, but there are some challenges storing two copies like data redundancy and second is any changes made in table has to be made in copies simultaneously i.e., synchronization.

But MySQL provides solution by providing temporary 2 copies of EMP table in Server RAM and you can join to itself it is known as SELF JOIN

```
select a.ename, b.ename from emp b, emp a
where a.mgr = b.empno;
```

- 2 copies of EMP table are opened in Server RAM, we are selecting a.ename from one table and b.ename from another table, So 'a' table contains 5 rows of ENAME and MGR column and 'b' table contains 5 rows of ENAME and EMPNO column
- 'a' is driving table and 'b' is driven table, MySQL will fetch the first row of 'a' table i.e. a.mgr = 4, then it will go to 'b' table and search for b.empno = 4 where condition is satisfied and it will return a.ename = 'Arun' and b.ename= 'Jack' in output and MySQL comes back to table 'a' and fetch second row(a.mgr = 1) then it will go to 'b' table and search for b.empno = 1, here where condition is satisfied and it will return a.ename = 'Ali' and b.ename= 'Arun' in output and comes back to table 'a' , and this cycle will continue till end of table. At end this temporary a and b table are dropped

- DO NOT SPECIFY AN ALIAS FOR TABLENAME UNNECESSARILY, BECAUSE THE MOMENT YOU SPECIFY AN ALIAS FOR TABLENAME, A COPY OF THE TABLE IS BROUGHT IN SERVER RAM; NOT ONLY WILL YOUR SELECT STATEMENT WILL BE SLOW, YOU WILL SLOW THE SELECT STATEMENT AND PROGRAMS OF OTHER USERS ALSO e.g.,

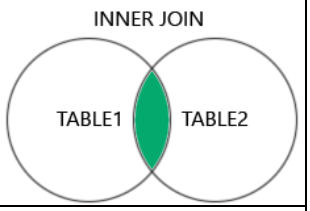
```
select dname, ename from emp e, dept d
where d.deptno = e.deptno;
```

- **YOU NEED TO SPECIFY AN ALIAS FOR TABLENAME IN THE RARE EVENT WHEN YOU WRITE A SELF-JOIN**

- **Cartesian join is the Fastest join, because there is no WHERE clause hence there is no searching involved**

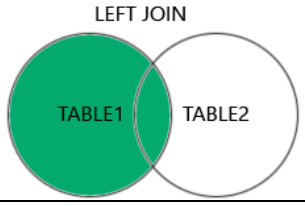
INNER JOIN: Returns records that have matching values in both tables

```
SELECT * FROM emp INNER JOIN dept
ON emp.DEPTNO = dept.DEPTNO;
```



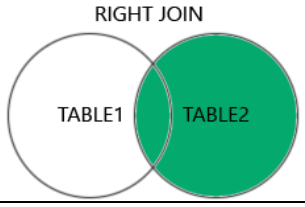
LEFT JOIN: Returns all records from the left table, and the matched records from the right table

```
SELECT dname, ename FROM emp LEFT JOIN dept
ON dept.deptno = emp.deptno;
```

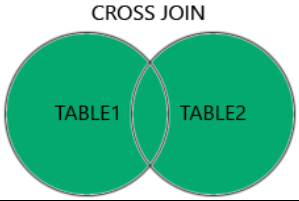


RIGHT JOIN: Returns all records from the right table, and the matched records from the left table

```
SELECT dname, ename FROM emp RIGHT JOIN dept
ON dept.deptno = emp.deptno;
```



CROSS JOIN: Returns all records from both tables



Joining 3 or more tables

EMP table					
EMPNO	ENAME	SAL	**DEPTNO**	JOB	MGR
1	Arun	8000	**1**	M	4
2	Ali	7000	**1**	C	1
3	Kirun	3000	**1**	C	1
4	Jack	9000	**2**	M	null
5	Thomas	8000	**2**	C	4

DEPT table		
DEPTNO	DNAME	LOC
1	TRN	Bby
2	EXP	Dlh
3	MKTG	Cal

DEPTHEAD	
DEPTNO	DHEAD
1	Arun
2	Jack

```
create table depthead(
deptno INT,
dhead VARCHAR(50),
);
INSERT INTO emp (deptno, dhead)
VALUES
(1, 'Arun'),
(2, 'Jack');
```

```
                                    (5)  (3)     (2)        ← ROWS
select dname, ename, dhead from emp, dept, depthead
where depthead.deptno = dept.deptno
and dept.deptno = emp.deptno;
```

OUTPUT

```
+-------+--------+-------+
| dname | ename  | dhead |
+-------+--------+-------+
| TRN   | Arun   | Arun  |
| TRN   | Ali    | Arun  |
| TRN   | Kirun  | Arun  |
| EXP   | Jack   | Jack  |
| EXP   | Thomas | Jack  |
+-------+--------+-------+
```

Types of relationship

- **1 : 1** **(Depthead : Dept) or (Dept : Depthead)**
- **1 : Many** **(Dept : Emp) and (Depthead : Emp)**
- **Many : 1** **(Emp : Dept) and (Emp : Depthead)**
- **Many : Many** **(Projects : Emp) or (Emp : Projects)**

PROJECTS table		
PROJNO	**CLIENTNAME**	**DESCRIP**
P1	Deloitte	CGS
P2	BNP Paribas	Macros Prg
P3	Morgan Stanley	AMS
P4	ICICI Bank	PPS
P5	AMFI	Website Dev

EMP table					
EMPNO	ENAME	SAL	**DEPTNO**	JOB	MGR
1	Arun	8000	**1**	M	4
2	Ali	7000	**1**	C	1
3	Kirun	3000	**1**	C	1
4	Jack	9000	**2**	M	null
5	Thomas	8000	**2**	C	4

PROJECTS_EMP	
pno	**empno**
P1	1
P1	2
P1	4
P2	1
P2	3
P3	2
P3	4
P3	5

INTERSECTION TABLE is required for Many : Many relationship

```
select clientname, ename from projects_emp, emp, projects
where projects.projno = projects_emp.projno
and emp.empno = projects_emp.empno
order by 1,2;
```

DAY 8

MySQL – SQL – SUB- QUERIES (V. imp)

- Nested queries (Query within Query) (SELECT within SELECT)

EMP table

EMPNO	ENAME	SAL	DEPTNO	JOB	MGR
1	Arun	8000	1	M	4
2	Ali	7000	1	C	1
3	Kirun	3000	1	C	1
4	Jack	9000	2	M	null
5	Thomas	8000	2	C	4

```
select min(sal) from emp;
```

```
+----------+
| min(sal) |
+----------+
|     3000 |
+----------+
```

Display the ENAME who is receiving <u>min(sal) :-</u>

```
select ename, min(sal) from emp;          ERROR
```

- Above SELECT statement will not work! you will get error because **you cannot select column of table (ename) along with a group function**

```
select ename from emp where sal = min(sal);     ERROR
```

- Above SELECT statement will not work! you will get an error because **you cannot use group function in WHERE clause** because WHERE clause is used for searching, searching takes place inside the Server HD. At time of searching inside the table min(sal) does not exist, min(sal) is calculated by MySQL after the rows are brought in Server RAM

If you want to solve the above problem, you have to break the SELECT statement in two steps:

First find out what is minimum sal, then find out the ename corresponding to the value of min(sal)

Single Row Subquery

Subqueries that return a single row as an output to their parent query are called single-row subqueries.

Single-row subqueries are used in a SQL SELECT statement with HAVING clause, WHERE clause, or a FROM clause and a comparison operator. Single-row subqueries are used in the SELECT statement.

Display the ENAME who is receiving min(sal) :-

```
select ename from emp ◄—— Main query (Parent query) (Outer query)
where sal =
(select min(sal) from emp);◄—— Sub-query (Child query) (Inner query)
```

```
+-------+
| ename |
+-------+
| Kirun |
+-------+
```

```
select ename from emp where sal = (select min(sal) from emp);
```

- Execution is from top to bottom, left to right. since anything in () bracket has higher priority, so first sub-query (select min(sal) from emp) will execute first and return minimum sal as 3000, but this value 3000 returned by Sub-query won't be displayed on the screen rather it would go as input/ parameter to the main query i.e.,

```
select ename from emp where sal = 3000
```

- Then main query will execute and give name of employee (i.e., Kiran) who is getting minimum sal = 3000 in the output

```
select ename from emp
where sal =
(select min(sal) from emp
where deptno =
(select.....));
```

- max up to 255 levels for sub-queries
  (This limit of SQL can be exceeded with help of **Views**)
- **JOIN IS FASTER THAN SUB-QUERY** because when you solve the problem using Join you solve the problem using one SELECT statement, whereas when you use sub-queries, you require 2 or more SELECT statements
- The more the number of SELECT statements, the slower it will be.

Display the 2nd largest sal :-

```
select max(sal) from emp
where sal <
(select max(sal) from emp);    ← Sub-query
```

```
+----------+
| max(sal) |
+----------+
|     8000 |
+----------+
```

```
select max(sal) from emp where sal < (select max(sal) from emp);
    Execute 2nd                  Execute 1st
select max(sal) from emp where sal < (9000);
```

rows less 9000 are retrieved from this, max(sal) i.e., 8000 is returned

8000
7000
3000
8000

- Statement (select max(sal) from emp); will execute first(because anything in ( ) bracket has higher priority) and return maximum sal i.e., 9000 which will not be seen on the screen but act as input / parameter to main query i.e.
  `select max(sal) from emp where sal < 9000`
- where sal < 9000 will execute and return all the rows less than 9000, and from these row, max(sal)/ highest row is selected which is 8000 and will be displayed in the output

EMP table					
EMPNO	ENAME	SAL	DEPTNO	JOB	MGR
1	Arun	8000	1	M	4
2	Ali	7000	1	C	1
3	Kirun	3000	1	C	1
4	Jack	9000	2	M	null
5	Thomas	8000	2	C	4

Display all the rows with same DEPTNO as 'Thomas':

```
select * from emp
where deptno =
(select deptno from emp
where ename = 'Thomas');
```

← Sub-query

Approach:

- First find out deptno of 'Thomas'

```
(select deptno from emp where ename = 'Thomas');
```

- From above command once you know the deptno of 'Thomas', now

```
select * from emp where deptno = (2)
```

- will show all the rows with same deptno as 'Thomas'

```
+-------+--------+------+--------+------+------+
| EMPNO | ENAME  | SAL  | DEPTNO | JOB  | MGR  |
+-------+--------+------+--------+------+------+
|     4 | Jack   | 9000 |      2 | M    | NULL |
|     5 | Thomas | 8000 |      2 | C    |    4 |
+-------+--------+------+--------+------+------+
```

```
select * from emp where deptno = (select deptno from emp where ename = 'Thomas');
          Execute 2nd                    Execute 1st
```

- First sub-query → `(select deptno from emp where ename = 'Thomas');` will execute and return value 2, the value 2 which is been returned will not been displayed on the screen rather it goes as input/parameter to main-query
  `select * from emp where deptno = (2)`
- Main-query is executed and will show all the rows with same DEPTNO as 'Thomas'.

EMP table					
EMPNO	ENAME	SAL	DEPTNO	JOB	MGR
1	Arun	8000	1	M	4
2	Ali	7000	1	C	1
3	Kirun	3000	1	C	1
4	Jack	9000	2	M	null
5	Thomas	8000	2	C	4

Display all the rows who are doing the same JOB as 'Kirun':

```
select * from emp
where job =
(select job from emp where ename = 'Kirun');
```

```
+-------+--------+------+--------+------+------+
| EMPNO | ENAME  | SAL  | DEPTNO | JOB  | MGR  |
+-------+--------+------+--------+------+------+
|     2 | Ali    | 7000 |      1 | C    |    1 |
|     3 | Kirun  | 3000 |      1 | C    |    1 |
|     5 | Thomas | 8000 |      2 | C    |    4 |
+-------+--------+------+--------+------+------+
```

Approach:

- First find out job of 'Kirun'

```
(select job from emp where ename = 'Kirun');
```

- From above command once you know the job of 'Kirun', now

```
select * from emp
where job =(C)
```

- will show all the rows same job as 'Kirun'

Execution

```
select * from emp where job = (select job from emp where ename = 'Kirun');
  main-query→Execute 2nd                    sub-query→Execute 1st
```

- First the sub-query will execute `(select job from emp where ename = 'Kirun');` and this will return value 'C', this value will act as input / parameter for main-query i.e.

```
select * from emp where job = (C)
```

- Main-query is executed and will show all the rows with same JOB as 'Kirun'

Using sub-query with DML commands:

In other RDMBS: -

```
delete from emp
where deptno =
(select deptno from emp
where ename = 'Thomas');
```

```
update emp set sal = 1000
where job =
(select job from emp
where ename = 'Kirun');
```

- **ABOVE 2 COMMANDS ARE NOT SUPPORTED BY MySQL**

- **In MySQL you cannot UPDATE or DELETE a table from which you are currently selecting (it creates a problem in Read and Write consistency)**
- **Using sub-query with DML commands: -**
  Solution

```
delete from emp
where deptno =
(select abcd.deptno from
(select deptno from emp
where ename = 'Thomas')abcd);
```

☒ **Innermost Subquery Execution**:

```
select deptno from emp where ename = 'Thomas';
```

- This subquery searches the emp table to find the department number (deptno) where the employee's name (ename) is 'Thomas'.
- The database engine executes this query and retrieves the department number associated with the employee 'Thomas'.

Example: Suppose the result of this subquery is 2 (meaning 'Thomas' is in department 2)

☒ **Subquery Result Assignment**:

```
(select abcd.deptno from (select deptno from emp where ename =
'Thomas') abcd)
```

- Here, the result of the inner subquery (`select deptno from emp where ename = 'Thomas'`) is assigned an alias abcd.

- This assigns an alias abcd to the result of the innermost subquery. So, abcd now represents a derived table (temporary result set) containing one column the department number (deptno) and one row containing 2 of the employee named 'Thomas'.

Example: If 'Thomas' is in department 2, then abcd.deptno would effectively be 2.

☒ **Main Delete Query Execution**:

```
delete from emp where deptno = (select abcd.deptno from (select
deptno from emp where ename = 'Thomas') abcd);
```

- Now, the outer delete query executes:
  - It attempts to delete rows from the emp table where the deptno matches the result of the subquery (`select abcd.deptno from ...`).
  - This subquery (`select abcd.deptno from ...`) refers to the temporary result set abcd which contains the department number (`deptno`) of 'Thomas'.

Example Execution: Given our example, the delete query effectively becomes:

```
delete from emp where deptno = 2;
```

- This means it will delete all rows from the emp table where the deptno column equals 2.

☒ **Deletion Process**:

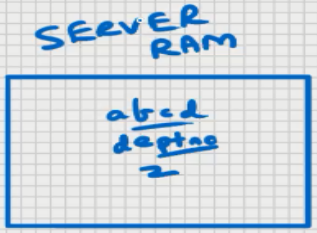
- The database engine then iterates through each row in the emp table.
- For each row, it checks if the deptno matches 2 (the department number obtained from the subquery).
- If a match is found, that particular row is deleted from the emp table.

☒ **Final Outcome**:

- After the delete operation completes, all rows in the emp table where the department number matches the department number of the employee named 'Thomas' are removed from the table.

EMP

EMPNO	ENAME	SAL	DEPTNO	JOB	MGR
1	Arun	8000	1	M	4
2	Ali	7000	1	C	1
3	Kirun	3000	1	C	1
4	Jack	9000	2	M	
5	Thomas	8000	2	C	4




```
update emp set sal = 1000
where job =
(select abcd.job from
(select job from emp
where ename = 'Kirun')as abcd);
```

- Innermost Subquery Explanation:

`'(select job from emp where ename = 'Kirun') as abcd'`: This subquery retrieves the '`job`' of the employee whose name is 'Kirun' and aliases the result set as '`abcd`'.

- Middle Subquery Explanation:

`'select abcd.job from ...':` This subquery selects the '`job`' from the aliased subquery '`abcd`'.

- Main `UPDATE` Statement:

`'update emp set sal = 1000 where job = ...':` This is the main '`UPDATE`' statement.
`'where job = ...'`: Specifies that only rows from the `emp` table where the '`job`' column matches the '`job`' retrieved from the subquery (`select abcd.job ...`) will be updated.

Single Row Subquery

- Subqueries that return a single row as an output to their parent query are called single-row subqueries.
- Single-row subqueries are used in a SQL SELECT statement with HAVING clause, WHERE clause, or a FROM clause and a comparison operator.

Multi-Row Sub-query

(Sub-query returns multiple rows)

Subqueries that return multiple rows as an output to their parent query are called multiple-row subqueries.
Multiple row subqueries can be used in a SQL SELECT statement with a HAVING clause, WHERE clause, a FROM clause, and a logical operator (ALL, IN, NOT IN, and ANY)

EMP

EMPNO	ENAME	SAL	DEPTNO	JOB	MGR
1	Arun	8000	1	M	4
2	Ali	7000	1	C	1
3	Kirun	3000	1	C	1
4	Jack	9000	2	M	
5	Thomas	8000	2	C	4

Display all the rows who are receiving the SAL equal to any one of manager

Solution 1: By using Sub-query

```
select * from emp
where sal = any            (8000,9000)
(select sal from emp
where job = 'M');
```

```
+-------+--------+------+--------+------+------+
| EMPNO | ENAME  | SAL  | DEPTNO | JOB  | MGR  |
+-------+--------+------+--------+------+------+
|     1 | Arun   | 8000 |      1 | M    |    4 |
|     4 | Jack   | 9000 |      2 | M    | NULL |
|     5 | Thomas | 8000 |      2 | C    |    4 |
+-------+--------+------+--------+------+------+
```

- **any** operator that performs logical OR

Solution 2: By using JOIN

```
select * from emp
where sal in          (8000,9000)
(select sal from emp
where job = 'M');
```

```
+-------+--------+------+--------+------+------+
| EMPNO | ENAME  | SAL  | DEPTNO | JOB  | MGR  |
+-------+--------+------+--------+------+------+
|     1 | Arun   | 8000 |      1 | M    |    4 |
|     4 | Jack   | 9000 |      2 | M    | NULL |
|     5 | Thomas | 8000 |      2 | C    |    4 |
+-------+--------+------+--------+------+------+
```

- Since you want to check for equality IN operator is recommended

- **IN operator is faster than ANY operator because ANY operator is overloaded**
- **If you want to check for equality or inequality, then use IN operator**
- **If you want to check for greater than or less than, then use ANY operator**

```
select * from emp
where sal >=          (8000)
(select min(sal) from emp
where job = 'M');
```

```
+-------+--------+------+--------+------+------+
| EMPNO | ENAME  | SAL  | DEPTNO | JOB  | MGR  |
+-------+--------+------+--------+------+------+
|     1 | Arun   | 8000 |      1 | M    |    4 |
|     4 | Jack   | 9000 |      2 | M    | NULL |
|     5 | Thomas | 8000 |      2 | C    |    4 |
+-------+--------+------+--------+------+------+
```

To make it work faster: -

- JOIN is faster than sub-query; first preference should always be to solve the problem using a JOIN
- Try to reduce the number of levels of Sub-queries
- Try to reduce the number of rows returned by Sub-query

Assumption, 3rd row is 13000

EMP table					
EMPNO	ENAME	SAL	DEPTNO	JOB	MGR
1	Arun	8000	1	M	4
2	Ali	7000	1	C	1
3	Kirun	13000	1	C	1
4	Jack	9000	2	M	null
5	Thomas	8000	2	C	4

Display the rows who are receiving a SAL greater than all of the Manager: -

```
select * from emp
where sal > all           (8000,9000)
(select sal from emp
where job = 'M');
```

- **all** is special operator performs logical AND

1. Subquery Execution:

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

This subquery fetches the salaries of employees whose job is 'M'.
Result of subquery:

SAL
8000
9000

2. Main Query Execution:

```
SELECT * FROM emp WHERE sal > ALL (SELECT sal FROM emp WHERE job = 'M');
```

- The `'ALL'` operator ensures that the `'sal'` value in the main query must be greater than all values returned by the subquery.
- The condition `'sal > ALL (8000, 9000)'` means that the `'sal'` in the main query must be greater than `'9000'.`

3. Comparison and Result:

- Comparing each employee's salary against 9000:
- Arun: 8000 (not greater than 9000)
- Ali: 7000 (not greater than 9000)
- Kirun: 13000 (greater than 9000)
- Jack: 9000 (not greater than 9000)
- Thomas: 8000 (not greater than 9000)
- The only employee with a salary greater than 9000 is Kirun.

Therefore, the query will return:

EMPNO	ENAME	SAL	DEPTNO	JOB	MGR
3	Kirun	13000	1	C	1

This result set includes Kirun, whose salary (13000) is greater than the highest salary (9000) of employees with the job title 'M'.

ANY → Logical OR
IN → Logical OR
ALL → Logical AND

```
select * from emp
where sal >           (9000)
(select max(sal) from emp
where job = 'M');
```

Assumption, 3rd row is 3000

EMP table					
EMPNO	ENAME	SAL	**DEPTNO**	JOB	MGR
1	Arun	8000	**1**	M	4
2	Ali	7000	**1**	C	1
3	Kirun	3000	**1**	C	1
4	Jack	9000	**2**	M	null
5	Thomas	8000	**2**	C	4

DEPT table		
DEPTNO	DNAME	LOC
1	TRN	Bby
2	EXP	Dlh
3	MKTG	Cal

Using Sub-query in the HAVING clause: -

Display the department name i.e., DNAME that is having max(sum(sal)): -

```
select deptno,sum(sal) from emp
group by deptno;
```

OUTPUT

deptno	sum(sal)
1	18000
2	17000

```
select sum(sal) from emp
group by deptno;
```

OUTPUT

sum(sal)
18000
17000

```
select max(sum(sal)) from emp
group by deptno;
```

OUTPUT

sum(sal)
18000

```
select sum(sal) sum_sal from emp
group by deptno;
```

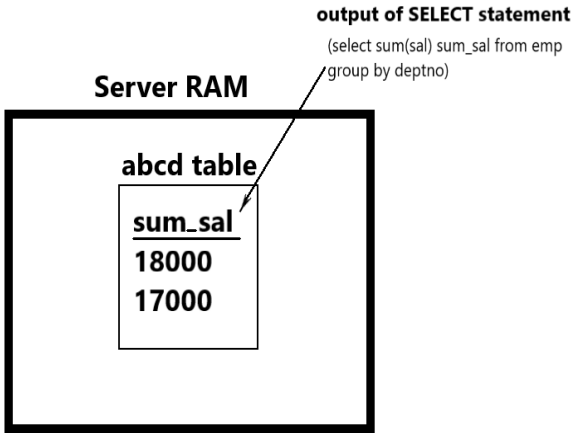
OUTPUT

sum_sal
18000
17000

```
select max(sum_sal) from
(select sum(sal) sum_sal from emp
group by deptno) abcd;
```

OUTPUT

max(sum_sal)
18000




```
select deptno, sum(sal) from emp
group by deptno
having sum(sal) =
(select max(sum_sal) from
(select sum(sal) sum_sal from emp
group by deptno) abcd);
```

OUTPUT

deptno	sum(sal)
1	18000

```
select dname, sum(sal) from emp, dept
where dept.deptno = emp.deptno
group by dname
having sum(sal) =
(select max(sum_sal) from
(select sum(sal) sum_sal from emp
group by deptno) abcd);
```

OUTPUT

dname	sum(sal)
TRN	18000

In ORACLE: -

```
select dname, sum(sal) from emp, dept
where dept.deptno = emp.deptno
group by dname
having sum(sal) =
(select max(sum(sal) from emp
group by deptno);
```

OUTPUT

dname	sum(sal)
TRN	18000

Correlated Sub-query (using the EXISTS operator)

(Note: This topic is explained in most of books, w3school website)

- This is the exception when Sub-query is faster than Join
- if you have a Join along with Distinct, to make it work faster, use Correlated Sub-query (use EXISTS operator)

Display the DNAMES that contain employees: -

Solution 1:-

```
select deptno from emp;
```

```
+--------+
| deptno |
+--------+
|      1 |
|      1 |
|      1 |
|      2 |
|      2 |
+--------+
```

```
select distinct deptno from emp;
```

```
+--------+
| deptno |
+--------+
|      1 |
|      2 |
+--------+
```

```
select dname from dept
where deptno = any        (1,2)
(select distinct deptno from emp);
```

```
+-------+
| dname |
+-------+
| TRN   |
| EXP   |
+-------+
```

```
select dname from dept
where deptno in           (1,2)
(select distinct deptno from emp);
```

```
+-------+
| dname |
+-------+
| TRN   |
| EXP   |
+-------+
```

```
select dname from dept
where deptno not in       (1,2)
(select distinct deptno from emp);
```

```
+-------+
| dname |
+-------+
| MKTG  |
+-------+
```

Solution 2: -

```
select dname from emp, dept
where dept.deptno = emp.deptno;
```

```
+-------+
| dname |
+-------+
| TRN   |
| TRN   |
| TRN   |
| EXP   |
| EXP   |
+-------+
```

```
select distinct dname from emp, dept
where dept.deptno = emp.deptno;
```

```
+-------+
| dname |
+-------+
| TRN   |
| EXP   |
+-------+
```

Solution 3:-

```
select dname from dept where exists
(select deptno from emp
where dept.deptno = emp.deptno);
```

```
+-------+
| dname |
+-------+
| TRN   |
| EXP   |
+-------+
```

```
select dname from dept where exists ──────→ Main query
   TRUE/FALSE
(select deptno from emp
where dept.deptno = emp.deptno); } Sub query
```

- First the main query is executed
- For every row returned by main query, it will run the sub- query once
- The sub-query returns a boolean TRUE value or FALSE value
- If sub-query returns a TRUE value, then the main query is eventually executed for that row
- If sub-query returns a FALSE value, then the main query is not executed for that row
- Unlike earlier, you do not use DISTINCT here, hence no sorting takes place in Server RAM; these speeds it up
- Unlike a traditional join, the number of full table scans is reduced; these further speeds it up

MySQL-SQL-Set Operators

- Dr. Codd (Mathematician/statistician)→founder of RDBMS (1968)
- based on Set theory

EMP1 table	
empno	ename
1	A
2	B
3	C

EMP2 table	
empno	ename
1	A
2	B
4	D
5	E

```
select empno, ename from emp1
union
select empno, ename from emp2;
```

```
+-------+-------+
| empno | ename |
+-------+-------+
|     1 | A     |
|     2 | B     |
|     3 | C     |
|     4 | D     |
|     5 | E     |
+-------+-------+
```

UNION → It will combine the output of both the SELECT statements and it will suppress the duplicates

- Structures of both the SELECT statements has to match
  (Number of columns and corresponding datatypes has to match)
  (Column names may be different, in the output the column heading is taken from first SELECT statement not second SELECT statement therefore if you want to specify ALIAS, then specify ALIAS in first SELECT statement)

EMP1 table	
empno1	ename
1	A
2	B
3	C

EMP2 table	
empno2	ename
1	A
2	B
4	D
5	E

```
select empno1, ename from emp1
union
select empno2, ename from emp2;
```

```
+--------+-------+
| empno1 | ename |
+--------+-------+
|      1 | A     |
|      2 | B     |
|      3 | C     |
|      4 | D     |
|      5 | E     |
+--------+-------+
```

```
select empno1, ename from emp1
union
select empno2, ename from emp2
order by 1;
```

```
+--------+-------+
| empno1 | ename |
+--------+-------+
|      1 | A     |
|      2 | B     |
|      3 | C     |
|      4 | D     |
|      5 | E     |
+--------+-------+
```

```
select empno1, ename from emp1
union all
select empno2, ename from emp2
order by 1;
```

```
+--------+-------+
| empno1 | ename |
+--------+-------+
|      1 | A     |
|      1 | A     |
|      2 | B     |
|      2 | B     |
|      3 | C     |
|      4 | D     |
|      5 | E     |
+--------+-------+
```

UNION ALL → It will combine the output of both the SELECT statements and the duplicates are not suppressed

```
select empno1, ename from emp1
intersect
select empno2, ename from emp2
order by 1;
```

```
+--------+-------+
| empno1 | ename |
+--------+-------+
|      1 | A     |
|      2 | B     |
+--------+-------+
```

INTERSECT → It will return what is common in both the SELECT statements and the duplicates are suppressed

```
select empno1, ename from emp1
except
select empno2, ename from emp2
order by 1;
```

```
+--------+-------+
| empno1 | ename |
+--------+-------+
|      3 | C     |
+--------+-------+
```

EXCEPT → what is present in the first SELECT statement, and not present in the second SELECT statement, and the duplicates are suppressed

```
select.....
    union
select.....
    except
select.....
    union all
select.....
    intersect
select.....
    union
select.....
    order by x;
```

- max upto 255 SELECT statements
  (this limit of SQL can be exceeded with help of Views)

Pseudo Columns

- Pseudo columns are fake columns (virtual columns)
- It is not a column of the table, but you can use it in SELECT statement
  e.g.,
  a. Computed columns (annual =sal*12)
  b. Expressions (net_earnings = sal + comm)
  c. Function-based columns
     (avg_sal = avg(sal), u_ename = upper(ename), r_sal = round(sal, -3))

RDBMS supplied Pseudo columns: -

ROWID

- Rowid stands for row identifier
- Rowid is the row address
- Rowid is the actual physical memory location in the DB Server HD where that row is stored
- Rowid is an encrypted fixed-length string of 18 characters
- When you SELECT from a table, the order of rows in the output will always be in ascending order of Rowid
- No two rows of any table in the DB Server HD can have the same Rowid
- Rowid works as a Unique identifier for every row in the DB Server HD
- Rowid is the system used by the RDBMS to distinguish between 2 rows in the DB Server HD
- When you UPDATE a row, if the row length is increasing, the Rowid may change
- You can use Rowid to UPDATE or DELETE the duplicate rows

```
select rowid, ename, sal from emp;

ROWID              ENAME             SAL
------------------ ---------- ----------
AAAS1HAABAAAdzxAAA KING              5000
AAAS1HAABAAAdzxAAB BLAKE             2850
AAAS1HAABAAAdzxAAC CLARK             2450
AAAS1HAABAAAdzxAAD JONES             2975
AAAS1HAABAAAdzxAAE MARTIN            1250
AAAS1HAABAAAdzxAAF ALLEN             1600
AAAS1HAABAAAdzxAAG TURNER            1500
AAAS1HAABAAAdzxAAH JAMES              950
AAAS1HAABAAAdzxAAI WARD              1250
```

Rowid is used internally by MySQL

1. To distinguish between 2 rows in the DB Server HD
2. Row Locking
3. To manage Indexes
4. To manage Cursors
5. Row management

- Rowid is available in Oracle and MySQL
- You can view the Rowid in Oracle
- You cannot view the Rowid in MySQL

```
select rowid, ename, sal from emp where rowid= 'AAAS1HAABAAAdzxAAA'
```

ALTER table (DDL command)

- Rename a table
- Add a column to the table
- Drop a column
- Increase width of column

Indirectly: -

- Reduce width of column
- Change datatype of column
- Copy rows from one table into another table
- Copy table (for testing purposes)
- Copy only the structure of table
- Rename a column
- Change a position of columns in the table structure
  (For storage considerations because of null values)

NOTE, For counting null values: `select count(*)- count(sal) from emp;`

EMP table		
empno	ename	sal
101	KING	5000
102	SCOTT	6000

RENAME TABLE: renames one or more tables.

SYNTAX:

```
RENAME TABLE old_table TO new_table;
```

Rename table emp to employees

```
rename table emp to employees;
```

- Rename is a DDL command (extra in MySQL and Oracle)

Note rename table is NOT AT ALL GOOD IDEA / NOT RECOMMENDED

You May Loose Your Job !!!!!! 

ALTER TABLE:

- The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.
- The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

Add a column to table

SYNTAX

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

```
alter table emp add gst float;
```

- By default, when you add a column, it gets added to end of table structure

NOTE:

select * from emp; (not recommended)
select empno, ename, sal from emp; (recommended)
insert into emp values(……); (not recommended)
insert into emp(empno,ename,sal) values(……); (recommended)

Drop a column from table

SYNTAX

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

```
alter table emp drop column gst;
```

Increase width of column i.e. varchar(25) to varchar(30)

SYNTAX

```
ALTER TABLE table_name
MODIFY column_name
varchar(new_length);
```

```
alter table emp modify ename varchar(30);
```

Reduce width of column i.e. varchar(25) to varchar(20)

In Oracle:

- You can reduce the width provided the contents are null

```
alter table emp add x varchar(25);

update emp set x = ename, ename = null;

alter table emp modify ename varchar(20);

/* Data testing with X column; check none of the ENAMES are
exceeding 20 characters */

update emp set ename = x;

alter table emp drop column x;
```

In MySQL:

```
alter table emp modify ename varchar(20); ← DATA WILL GET TRUNCATED
```

- It is recommended that you use the above discussed Oracle solution to reduce the width of column

Whenever you create a table, add few extra columns X1, X2…. for future purpose. These Extra Columns are known as Extension columns used to extend table

Change datatype of column

SYNTAX

```
ALTER TABLE table_name
MODIFY COLUMN column_name datatype;
```

```
alter table emp modify empno char(20);
```

Copy rows from one table into another table

DAY 9

EMP_K		
EMPNO	**ENAME**	**SAL**
101	KING	5000
102	SCOTT	6000

EMP_J		
EMPNO	**ENAME**	**SAL**
103	A	NULL
104	B	NULL
105	C	NULL

Copy rows from one table into another table: -

```
insert into emp_k
select * from emp_j;
```

To copy specific rows only: -

```
insert into emp_k
select * from emp_j where……;
```

Copy a table (for testing purposes): -

```
create table emp_copy
as
select * from emp;
```

- Above command is working in two steps, in first step it will create a table 'emp_copy' which bases the structure of SELECT statement, in second step SELECT statement is executed, the output of SELECT statement is not visible on screen it is inserted into 'emp_copy'
- Note: Grants and permission associated with emp table will not be copied

To copy specific columns/rows only:-

```
create table emp_copy
as
select empno, ename from emp;
```

copy specific columns only

```
create table emp_copy
as
select empno, ename from emp where ……;
```

copy specific rows only

Copy only the structure of table: -

(Create a new table emp2 whose structure is same as emp table, but don't want a data of emp table)

SOLUTION 1: -

```
create table emp2
as
select * from emp;

delete from emp2;

commit;
```

SOLUTION 2: -

```
create table emp2
as
select * from emp;

truncate table emp2;
```

- truncate will DELETE all the rows and COMMIT

Difference between DELETE AND TRUNCATE

	DELETE	TRUNCATE
1	DML command	DDL command
2	Requires commit	Auto Commit
3	ROLLBACK is possible	ROLLBACK is not possible
4	Can delete specific rows only with the help of WHERE clause	You cannot use WHERE clause with Truncate command, Truncate will delete all the rows and commit
5	Common for all RDBMS	Extra command in MySQL and Oracle
6	ANSI standard	Not an ANSI standard
7	Free space is not deallocated	Free space is deallocated

DELETE	TRUNCATE
Syntax: DELETE FROM Table_Name; Or Delete FROM Table_Name WHERE CONDITION;	Syntax: TRUNCATE TABLE_Table_Name;
DML Command	DDL Command
Can be used with WHERE Clause to remove the specified data.	Cannot be used with WHERE Clause. Thus, TRUNCATE removes all records from the table.
Removes the rows one by one and thus, is slower than TRUNCATE.	Removes all the rows at a time and therefore, much faster than DELETE.
Activates the triggers as the operations are logged on	Cannot activate a trigger as the operation does not log individual

	Truncate	Delete
1.	Truncate is a DDL command	Delete is a DML command
2.	**Syntax: TRUNCATE TABLE Table_Name**	**Syntax: DELETE FROM Table_Name WHERE Condition**
3.	We can't delete records based on any condition. Truncate removes all the records it I because we can't use where clause in Truncate command.	We can delete records based on condition using Where clause.
4.	Truncate reset Identity column value.	Delete never reset Identity column value.
5.	Truncate command will never activates Triggers.	Delete command will activates Triggers.
6.	Truncate command can't be used on Parent table if foreign key relationship is present on primary key column.	Delete can be used on Parent table if foreign key relationship is present on primary key column.
7.	Truncate is faster compare to DELETE.	DELETE is slower compare to TRUNCATE.
8.	Records removed by Truncate can't be rolled back until it is used in Explicit Transaction.	Records removed by Delete can be rolled back.
9.	Truncate doesn't log Individual row deletion.	Delete logs individual row deletion.
10.	If Transaction is done, means COMMITED, then we cannot rollback TRUNCATE command.	we can still rollback DELETE command from LOG files, as DELETE write records them in Log file in case it is needed to rollback in future from LOG files.
11.	TRUNCATE TABLE always locks the table (including a schema (SCH-M) lock) and page but not each row	When the DELETE statement is executed using a row lock, each row in the table is locked for deletion.

http://teachmesqlserver.blogspot.in/

Comparison Basis	DELETE	TRUNCATE
Definition	The delete statement is used to remove single or multiple records from an existing table depending on the specified condition.	The truncate command removes the complete data from an existing table but not the table itself. It preserves the table structure or schema.
Language	It is a DML (Data Manipulation Language) command.	It is a DDL (Data Definition Language) command.
WHERE	It can use the WHERE clause to filter any specific row or data from the table.	It does not use the WHERE clause to filter records from the table.
Permission	We need to have DELETE permission to use this command.	We need to have ALTER permission to use this command.
Working	This command eliminates records one by one.	This command deletes the entire data page containing the records.
Lock	It will lock the row before deletion.	It will lock the data page before deletion.

Table Identity	This command does not reset the table identity because it only deletes the data.	It always resets the table identity.
Transaction	It maintains transaction logs for each deleted record.	It does not maintain transaction logs for each deleted data page.
Speed	Its speed is slow because it maintained the log.	Its execution is fast because it deleted entire data at a time without maintaining transaction logs.
Trigger	This command can also activate the trigger applied on the table and causes them to fire.	This command does not activate the triggers applied on the table to fire.
Restore	It allows us to restore the deleted data by using the COMMIT or ROLLBACK statement.	We cannot restore the deleted data after using executing this command.
Indexed view	It can be used with indexed views.	It cannot be used with indexed views.
Space	The DELETE statement occupies more transaction space than truncate because it maintains a log for each deleted row.	The TRUNCATE statement occupies less transaction space because it maintains a transaction log for the entire data page instead of each row.

Suppose you have EMP table of 1 million rows which consumes 10 GB of space and if you delete it and commit it by command

```
delete from emp;
commit;
```

Even though we have deleted EMP table, 10 GB of **HD space is not deallocated,** it is not made available for other users and other tables; still the 10GB HD space is reserved for EMP table

instead, if you want to surrender(free) HD space

```
drop from emp;
```

Now 10 GB of **HD space is deallocated**; the 10GB HD space is not reserved for EMP table; the space is made available for other users and other tables

if you want to retain EMP table:-

```
create table emp.....;
```

and emp will occupy 0 Byte HD space

```
truncate table emp;
```

benefit of truncate command is that it will Delete all the rows and Commit and it will surrender the 10 GB of HD space.
The 10 GB of **HD space is deallocated**; the 10GB HD space is not reserved for EMP table; the space is made available for other users and other tables

SOLUTION 3:-

NOTE: This is best and easiest method

```
create table emp2
as
select * from emp where 1 = 2;
```

- Above command is working in two steps
- Step 1: it will create table EMP2 which bases on the structure of EMP table
- Step 2: select statement is executed, output of SELECT statement is nothing no rows are selected, and structure of EMP table is copied
- Give some impossible WHERE clause so that no rows will get copied for example in above statement `where 1 = 2`

Rename a column: - ☠

NOTE: In actual practise renaming column is WORST IDEA, YOU MAY LOOSE YOUR JOB

```
create table emp2
as
select empno, ename, sal salary
from emp;

drop table emp;

rename table emp2 to emp;
```

EMP

EMPNO	ENAME	SAL
101	KING	5000
102	SCOTT	6000

SALARY

EMP2

EMPNO	ENAME	SALARY

How to Rename a Column in MySQL: (www.geeksforgeeks)

To rename a column in MySQL use the ALTER TABLE Statement with the **CHANGE** or **RENAME clause.** Both Change and Rename can be used to change the name of the SQL table column, The only difference is that CHANGE can be utilized to alter the datatype of the column.

The Syntax for Renaming and Changing the Value of a Column in MySQL:

Syntax for Change Clause:

ALTER TABLE table_name
CHANGE old_column_name new_column_name datatype;

Syntax for Rename Clause:

ALTER TABLE table_name
RENAME COLUMN old_column_name TO new_column_name;

How to Rename Multiple Columns in MySQL

To rename multiple columns in MySQL, you can adjust the syntax to:

For CHANGE Clause:

ALTER TABLE table_name
RENAME COLUMN old_column_name1 TO new_col_name1,
RENAME COLUMN old_column_name2 TO new_col_name2,
RENAME COLUMN old_column_name3 TO new_col_name3;

For RENAME Clause:

ALTER TABLE table_name
CHANGE old_column_name1 new_col_name1 Data Type,
CHANGE old_column_name2 new_col_name2 Data Type,
CHANGE old_column_name3 new_col_name3 Data Type;

Choosing Between CHANGE and RENAME

Both **CHANGE** and **RENAME** are SQL commands used to modify the name of a column in a table, but they serve different purposes and are used in different scenarios:

If you need to change **both the name and data type of a column** or modify other properties along with the name change, then CHANGE is more appropriate.

If you **only need to rename the column** without altering its data type or other attributes, then RENAME is the preferred choice.

Change a position of columns in the table structure :-

(For storage considerations because of null values)

```
create table emp2
as
select ename, sal, empno
from emp;

drop table emp;

rename table emp2 to emp;
```

MySQL – SQL – INDEXES (V. imp)

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

- Indexes are present in all RDBMS, all DBMS, and some of the programming language (C programming, C++, COBOL, Java) also
- To speed up search operations (for faster access)
- To speed up SELECT statement with a WHERE clause
- Indexes are automatically invoked by MySQL as and when required
- Indexes are automatically updated by MySQL for all the DML operations
- No upper limit on the number of indexes per table
  (You can create as many as you want)
- Larger the number of indexes, the slower would be the DML (INSERT, UPDATE, DELETE) operations
- Cannot index TEXT and BLOB columns
- Duplicate values are stored in an index
- Null values are not stored in an index
- If you have multiple independent columns in WHERE clause, then you should create a separate index for each column, MySQL will use the necessary indexes as and when required

EMP				
ROWID	**EMPNO**	**ENAME**	**SAL**	**DEPTNO**
X001	5	A	5000	1
X002	4	A	6000	1
X003	1	C	7000	1
X004	2	D	9000	2
X005	3	E	8000	2

ROWID is not column of table, it's a pseudo column (fake column)
In ORACLE we can see ROWID, In MySQL we cannot see ROWID

```
select * from emp where empno = 1;
```

How this command works internally?

- First MySQL is going to **read** SELECT statement, then it's going to **compile** SELECT statement, then it is going to make a **plan** how to execute SELECT statement, then it **follows** the plan
- Now for execution, MySQL will go to database→ it will go to Server HD→It will go and search inside EMP table and it is going to search each and every row for EMPNO = 1→When EMPNO = 1 is found MySQL is going to return the row

(MySQL is going to do full table scan to find rows where `empno = 1`. The matching rows are retrieved.)

- Although this method of finding matching rows is effective, it can become slow and resource-intensive as the size of the table increases. As a result, this approach may not be suitable for large tables having millions of rows or queries requiring frequent or rapid data access. You might experience high CPU usage on your database server, slow response times, and ultimately, dissatisfied users

In other RDBMS,

```
use index ind_empno;
select * from emp where empno = 1;
```

IND_EMPNO	
ROWID	EMPNO
X003	1
X004	2
X005	3
X002	4
X001	5

What happens internally?

When you type SELECT statement, the system will read→ compile→ plan→ execute,

Step 1→ **Read**: First of it is going to read `select * from emp where empno = 1;` Moment it encounters semicolon (;) it will stop reading

Step 2→ **Compile**: It is going to compile SELECT statement i.e., it will convert SELECT statement to machine language

Step 3→ **Plan**: In this step it is going to make a plan, how it going to execute SELECT statement, and plan will be as follows

1. MySQL is going to check if EMP table is present or not
   (Internally it is going to do *'show tables'* it will check, whether EMP table is present or not, if EMP table is present proceed further if not give an error message 'invalid table name')
2. If EMP table is present, it going to check whether EMP table is having empno column or not
   (Internally it is going to *describe EMP table,*
   it will check whether EMP table has empno column or not,
   if empno column is present proceed further;
   if empno column is not present give an error message 'invalid column name')
3. If empno column is present, it going to check whether user has permission on EMP table,
   (If user has permission, proceed further;
   if user has no permission, then give an error message 'insufficient privileges'
4. If user has permission, it going to check whether index is available for empno
   (If index is present, it goes and search inside index;
   If index is not present it will go inside EMP table and will do Full Table Scan)

Step 4→ **Execute**: It will follow above execution plan

EXECUTION PLAN: Plan created by MySQL as to how it is going to execute SELECT statement

In other RDBMS,

INSERT/UPDATE/DELETE

REINDEX;

```
select * from emp where empno = 1;
```

IND_EMPNO

ROWID	EMPNO
X003	1
X004	2
X005	3
X002	4
X001	5

creating index for empno column

```
select * from emp where ename = 'C';
```

IND_ENAME

ROWID	ENAME
X001	A
X002	A
X003	C
X004	D
X005	E

creating index for ename column

```
select * from emp where sal > 7000;
```

IND_SAL

ROWID	SAL
X001	5000
X002	6000
X003	7000
X005	8000
X004	9000

creating index for sal column

Here above we have made three different indexes for empno, ename, sal column; There is no upper limit on the number of indexes per table (you can create as many as you want)

Indexes are used to find rows with specific column values quickly. Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows. The larger the table, the more these costs. If the table has an index for the columns in question, MySQL can quickly determine the position to seek to in the middle of the data file without having to look at all the data. This is much faster than reading every row sequentially.

COMPOSITE INDEX → combine 2 or more inter-dependent columns in a single index

INDEX KEY→ column or set of columns on whose basis the index has been created

	EMP			
ROWID	EMPNO	ENAME	SAL	DEPTNO
X001	1	A	5000	1
X002	2	A	6000	1
X003	3	C	7000	1
X004	1	D	9000	2
X005	2	E	8000	2

```
select * from emp
where deptno = 1 and empno = 1;
```

IND_DEPTNO_EMPNO		
ROWID	DEPTNO	EMPNO
X001	1	1
X002	1	2
X003	1	3
X004	2	1
X005	2	2

INDEX KEY

SECONDARY INDEX KEY

PRIMARY INDEX KEY

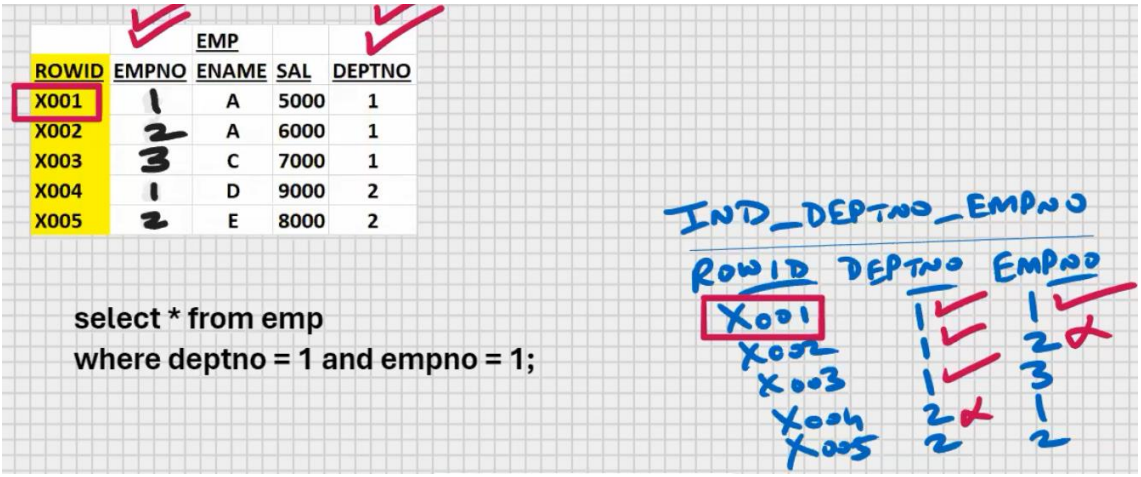
- Order is significant in composite index
  (It is recommended to keep parent column as first column and child column as second column, grandchild column last e.g., country, state, city)

In above table empno column is related to deptno, if you want to search specific row then you should search empno and deptno of department, because same empno is present in every department therefore to search specific row: -

```
select * from emp
where deptno = 1 and empno = 1;
```

To work above command faster combine both the column deptno and empno into single index ind_deptno_empno; in this index is going to store deptno in ascending order within that it is going to store corresponding empno in ascending order; it is going to store corresponding ROWID, after creating index above command will work faster

How compsite index works ?



Inside this index, MySQL will first search deptno = 1 when deptno = 2 is found in deptno column it will stop searching and then for these 3 rows it will search for empno = 1 and when 2 is found in empno column it will stop searching and whatever might be rowid it will go to that address and it is going to retain details very fast

Conditions when an index should be created

1. **SELECT statement with a WHERE clause, ORDER BY clause, DISTINCT, GROUP BY clause, UNION, INTERSECT, EXCEPT**

```
select * from emp where empno = 1;
```

- If you want above SELECT statement to work faster you need to create index IND_EMPNO for empno in advance

IND_EMPNO	
ROWID	EMPNO
X003	1
X004	2
X005	3
X002	4
X001	5

Here are some other SELECT statements

```
select * from emp order by empno;
```

```
select distinct empno from emp;
```

```
select deptno, sum(sal) from emp
group by deptno;
```

2. **If SELECT statement retrieves < 25 % of table data**

```
select * from emp where empno = 1;
select * from emp where empno = 5;
select * from emp where empno < 2;
```

retrieves less than 25 % of table data, so SELECT statement is faster

```
select * from emp where empno > 1;
```

- Above SELECT statement will work slower because index is available for empno, MySQL use index and almost all the rows of emp table are coming in the output and if large amount of data is coming in output the database may decide to ignore the index and perform a full table scan. This is because scanning the index to find large number of rows and then accessing the actual data rows can be more resource-intensive than simply scanning the entire table once.
- When a query retrieves more than 25% of the rows, the database might find it more efficient to perform a full table scan rather than using the index. This is because the overhead of using the index to access a large number of rows can exceed the cost of a full table scan.

3. **PRIMARY KEY columns and UNIQUE columns should always be indexed**

   Primary key column is best column for searching because primary key column does not contain duplicates so it will retrieve only one row

1. **Common column in Join operation should always be indexed**

EMP

ROWID	EMPNO	ENAME	SAL	DEPTNO
X001	1	A	5000	1
X002	2	A	6000	1
X003	3	C	7000	1
X004	4	D	9000	2
X005	5	E	8000	2

DEPT

ROWID	DEPTNO	DNAME	LOC
Y011	1	TRN	Bby
Y012	2	EXP	Dlh
Y013	3	MKTG	Cal

```
select dname, ename from emp, dept
where dept.deptno = emp.deptno;        ← SLOW
```

I2

ROWID	DEPTNO
X001	1
X002	1
X003	1
X004	2
X005	5

I1

ROWID	DEPTNO
Y011	1
Y012	2
Y013	3

CREATING INDEX

- Index name should have maximum 30 characters
- Try to give out meaningful name like *i_tablename_columnname*
- Table name and index name cannot be same
- Once you create index it is permanent, it will occupy HD space
- No upper limit on creating indexes, you can create indexes as much as you want

```
create index indexname on table(column);
```

```
create index i_emp_empno on emp(empno);
```

I_EMP_EMPNO

ROWID	EMPNO
X001	1
X002	1
X003	1
X004	2
X005	5

```
create index i_emp_ename on emp(ename);
```

```
create index i_emp_sal on emp(sal);
```

If you want index in descending order

```
create index i_emp_empno on emp(empno desc);
create index i_orders_onum on orders(onum desc);
```

- Latest new orders will be stored first at the top, and older order will be shifted below

COMPOSITE INDEX

```
create index i_emp_deptno_empno on emp(deptno, empno);
```

- The order is significant here, first column deptno will be primary index key i.e., first column in an index, second column empno would be secondary index key i.e., second column in an index
- Order will affect performance, specify parent column as first column, child column as second column, and grandchild column will be third column

```
create index i_emp_deptno_empno on emp(deptno desc, empno);
create index i_emp_deptno_empno on emp(deptno desc, empno desc);
```

To see which all indexes are created for specific table:

```
show indexes from emp;
```

To see all indexes on all tables in the database:

```
use information_schema;
select * from statistics;
```

To drop the index:

```
drop index i_emp_empno on emp;
```

```
drop table emp;
```

If you drop the table, then the indexes for that table are dropped automatically

```
create table emp2
as
select * from emp;
```

If you create a table using sub-query, then the indexes on original table are not copied into the new table; if you want indexes on the new table, then you will have to create them manually

UNIQUE INDEX

```
create unique index i_emp_empno on emp(empno);
```

- Performs one extra function; it won't allow the user to INSERT duplicates values for EMPNO
- At the time of creating the unique index, if you already have duplicate values in EMPNO, then MySQL will not allow you to create the unique index
- You cannot create more than one index on the same column, unless it is combined with other columns

Types of Indexes:-

1. Normal Index
2. Composite Index
3. Unique Index
4. Cluster Index
5. Bitmap Index
6. Index Organized table
7. Index partitioning
8. Global and Local Indexes

MySQL – SQL – CONSTRAINTS (V. imp)

- Limitations/restrictions imposed on a table

EMP			
EMPNO	**ENAME**	**SAL**	**DEPTNO**
1	A	5000	1
2	A	6000	1
3	C	7000	1
4	D	9000	2
5	E	8000	2

PRIMARY KEY constraint

- Primary column
- Column or set of columns that uniquely identifies a row, e.g., EMPNO
- Duplicates values are not allowed (it has to be unique)
- Null values are not allowed (this is a mandatory column)
- It's recommended that every table should have a Primary key (it helps from a long-term perspective)
- Purpose of Primary key is row uniqueness (with the help of primary key you should be able to distinguish between 2 rows of a table)
- TEXT and BLOB cannot be Primary key
- Unique index is automatically created

A **MySQL Primary Key** is a unique column/field in a table that should not contain duplicate or NULL values and is used to identify each record in the table uniquely.

The role of the **primary key constraint** is to maintain the integrity of the database by preventing duplicate and null values in the key column. A table can only have one primary key, but it can be defined on one or more fields and that is called a **composite primary key**. We can use the **AUTO_INCREMENT** attribute on the primary key field so that a new value is automatically added to the column when we create a new row.

COMPOSITE PRIMARY KEY constraint

EMP			
EMPNO	**ENAME**	**SAL**	**DEPTNO**
1	A	5000	1
2	A	6000	1
3	C	7000	1
1	D	9000	2
2	E	8000	2

- Combine 2 or more inter-dependent columns together to serve the purpose of Primary Key
- In MySQL, you can combine up to 32 columns in a Composite Primary key
- If you declare a Composite Primary key, then the composite unique index is created automatically
- If you cannot identify primary key column, then you add an extra column to the table to serve the purpose of Primary key, such a key is known as **SURROGATE KEY**
- For surrogate key column, CHAR datatype is recommended (because it has fixed length, searching and retrieval will be very fast)
- **YOU CAN HAVE ONLY 1 PRIMARY KEY PER TABLE**

SURROGATE KEY → is not a constraint

SURROGATE KEY → is a definition

SURROGATE KEY → if you cannot identify Primary key in the table, then you add a column to the table to serve the serve the purpose of Primary key; and such a Primary Key which is not an original column of the table is known as Surrogate Key

CANDIDATE KEY → is not a constraint

CANDIDATE KEY → is a definition

CANDIDATE KEY → besides Primary key in the table, any other column in the table that can also serve the purpose of Primary key (e.g., pan card no., passport no.), is a good candidate for Primary Key is known as Candidate Key

It is good idea to have couple of candidate keys in your table, because in future if you ALTER your table and DROP the Primary Key Column, then your table is left without a Primary Key, in that case you can make 1 of your candidate key column as the new Primary Key

Primary Key VS Candidate Key

Comparison Basis	Primary Key	Candidate Key
Definition	It is a unique and non-null key to identify each table's records in a schema uniquely.	It is also a unique key to identify records in relation or table uniquely.
Basic	A table or relation can contain only one primary key.	A table or relation can have more than one candidate key.
NULL	Any column of a primary key cannot be NULL.	The column of a candidate can contain a NULL value.
Objective	It is the essential part of a table or relation.	It signifies which key can be used as a primary key.
Use	It can be used as a candidate key.	It may or may not be used as a primary key.
Specify	It is not mandatory to specify a primary key for any relation.	There cannot be a relationship without specifying the candidate key.
Example	Consider a table "**student**" with columns (roll_no., name, class, DOB, email, mobile). Here **roll_no** column can be a primary key for the relationship because it identifies the student's records uniquely.	The **roll_no, mobile,** and **email** columns can be candidate keys in the given table because they can uniquely identify student's records.

EMP			
EMPNO	**ENAME**	**SAL**	**DEPTNO**
1	A	5000	1
2	A	6000	1
3	C	7000	1
4	D	9000	2
5	E	8000	2

If we want to create only one primary key column into the table, use the below syntax:

SYNTAX 1: -	**SYNTAX 2: -**

`CREATE TABLE table_name (`
`  column1 datatype PRIMARY KEY,`
`  column2 datatype,`
`  ...`
`);`
	`CREATE TABLE table_name (`
	`  column1 datatype,`
	`  column2 datatype,`
	`  PRIMARY KEY (column1)`
	`);`

`create table emp (`
`empno char(4) primary key,`
`ename varchar(25),`
`sal float,`
`deptno int`
`);`
	`create table emp (`
	`empno char(4) not null,`
	`ename varchar(25),`
	`sal float,`
	`deptno int,`
	`primary key (empno)`
	`);`

```
-- Insert the data
INSERT INTO EMP (EMPNO, ENAME, SAL, DEPTNO) VALUES
('1', 'A', 5000, 1),
('2', 'B', 6000, 1),
('3', 'C', 7000, 1),
('4', 'D', 9000, 2),
('5', 'E', 8000, 2);
```

In MySQL Command line Client

`insert into emp values('5', 'F', 5000, 2);` ← ERROR

- Since 5 is already present in empno column, duplicate values are not allowed in primary key column, you will get an error

`insert into emp values(null, 'F', 5000, 2);` ← ERROR

- Primary keys must always have a value, so they cannot be NULL, NULL values are not allowed in primary key column, you will get an error

- All the constraints are at Server level; you can perform the DML operations (INSERT, UPDATE, DELETE) using MySQL Command Line Client, or MySQL Workbench, or Java, or MS .Net or any other front-end software, the constraints will always be valid
- This is known as Data Integrity
- Internally a constraint is a MySQL created function, it performs the validations (checking)

After creating a table, if you want to find out which constraints you have created: -

```
select * from information_schema.table_constraints;
```

After creating a table, if you want to find out what constraints are present in *'juhu'* schema:-

```
select * from information_schema.table_constraints
where table_schema = 'juhu';
```

After creating a table, if you want to find out which column is Primary Key: -

```
select * from information_schema.key_column_usage
where table_name = 'emp';
```

Unique index is automatically created: -

```
show indexes from emp;
```

To drop Primary key Constraints: -

Syntax

```
ALTER TABLE table_name DROP PRIMARY KEY;
```

Example

```
alter table emp drop primary key;
```

To add Primary key afterwards, to an existing table: -

- You can add a primary key in an already existing table by using the **ALTER TABLE** statement.
- If the table has already one primary key then you can't add another by using the ALTER TABLE statement.

Syntax

```
ALTER TABLE table_name ADD PRIMARY KEY (column_name);
```

Example

```
alter table emp add primary key(empno);
```

Composite Primary key

SYNTAX

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    PRIMARY KEY (column1, column2)
);
```

EXAMPLE

```
create table emp (
empno char(4),
ename varchar(25),
sal float,
deptno int,
primary key (deptno,empno)
);
```

```
show indexes from emp;
```

Add a composite key afterwards to the existing table

```
alter table emp add primary key (deptno, empno);
```

To drop Composite key Constraints

Syntax

```
ALTER TABLE table_name DROP PRIMARY KEY;
```

Example

```
alter table emp drop primary key;
```

Command to drop Primary key and Composite key are same

To change the Primary Key column, drop the existing Primary Key constraint, and add a new Primary key constraint for the desired column

Constraints are of 2 types: -

1. Column level constraint (specified on one individual column)
2. Table level constraint (specified on combination of two or more columns) (Composite) (has to be specified at the end of the structure)

NOT NULL constraint

- Null values are not allowed (similar to primary key)
- Duplicate values are allowed (unlike primary key)
- You can have any number of not null constraint per table (unlike primary key)
- TEXT and BLOB can be not null (unlike primary key)
- Unique index not created automatically (unlike primary key)
- You cannot have a composite not null constraint, and therefore this will always a column level constraint

```
create table emp (
empno char(4),
ename varchar(25) not null,
sal float not null,
deptno int
);
```

In MySQL, nullability is a feature of the datatype

To find out which column are not null: -

```
desc emp;
```

To drop the not null constraint: -

```
alter table emp modify ename varchar(25) null;
```

To add the not null constraint afterwards to an existing table: -

```
alter table emp modify ename varchar(25) not null;
```

Solution For candidate Key: -

not null constraint + unique index

UNIQUE constraint

- Duplicate values are not allowed (similar to primary key)
- Null values are allowed (unlike primary key)
- You can INSERT any number of null values, but no duplicate values
- TEXT and BLOB cannot be unique
- Unique index is created automatically
- Can combine up to 32 columns in a composite unique
- YOU CAN HAVE ANY NUMBER OF UNIQUE CONSTRAINTS PER TABLE (UNLIKE PRIMARY KEY)

EMP				
EMPNO	**ENAME**	**SAL**	**DEPTNO**	**MOBILE_NO**
1	A	5000	1	9420056512
2	A	6000	1	9420056513
3	C	7000	1	9420056514
4	D	9000	2	9420056515
5	E	8000	2	9420056516

```
create table emp (
empno char(4),
ename varchar(25),
sal float,
deptno int,
mob_no char(15) unique,        ← column level constraint
unique (deptno,empno)     ← table level constraint
);
```

After creating a table, if you want to find out which constraints you have created: -

```
select * from information_schema.table_constraints;
```

After creating a table, if you want to find out what constraints are present in *'juhu'* schema: -

```
select * from information_schema.table_constraints
where table_schema = 'juhu';
```

After creating a table, if you want to find out which column is Unique constraint: -

```
select * from information_schema.key_column_usage
where table_name = 'emp';
```

Unique index is automatically created

```
show indexes from emp;
```

Unique constraint is also an index so to drop it: -

```
drop index mob_no on emp;
drop index deptno on emp;
```

To add unique constraints afterward to an existing table: -

```
alter table emp add constraint u_emp_mob_no unique(mob_no);
constraint u_emp_mob_no → optional
select * from information_schema.table_constraints
where table_schema = 'juhu';
```

- Column level constraint can be specified at the table level, but a table composite constraint cannot be specified at column level
- Column level constraint can be specified at the table level, except for the not null constraint which is always a column level constraint, and therefore the syntax will not support you from specifying it at table level

Primary Key VS Unique Key

Criteria	Primary Key	Unique Key
Basic Function	The primary key uniquely identifies each record in the table.	The unique key serves as a unique identifier for records when a primary key is absent.
NULL Values	The primary key cannot store NULL values.	The unique key can store a null value, but only one NULL value is allowed.
Purpose	It ensures entity integrity.	It enforces unique data.
Index Creation	By default, the primary key creates a clustered index.	The unique key generates a non-clustered index.
Number of Keys	Each table can have only one primary key.	A table can have multiple unique keys.
Value Modification	You cannot modify or delete values in a primary key.	You can modify the values in a unique key.
Uses	It identifies specific records in the table.	It prevents duplicate entries in a column, except for a NULL value.

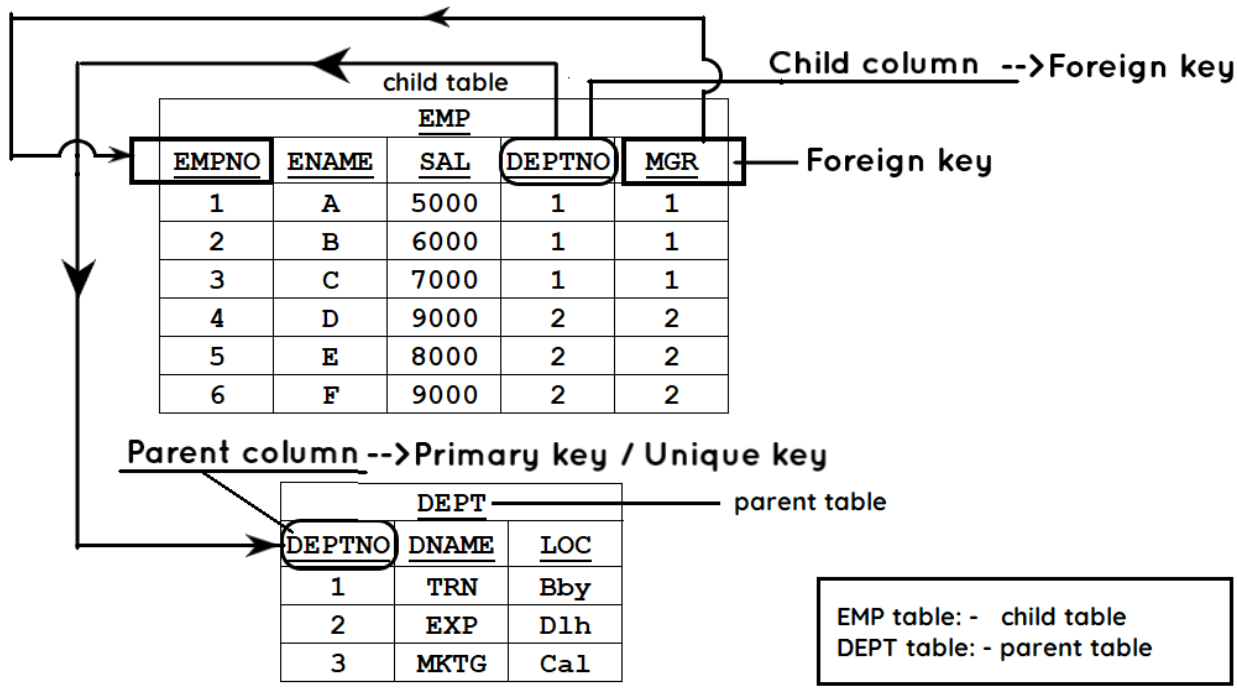
DAY 10

FOREIGN KEY constraint: -

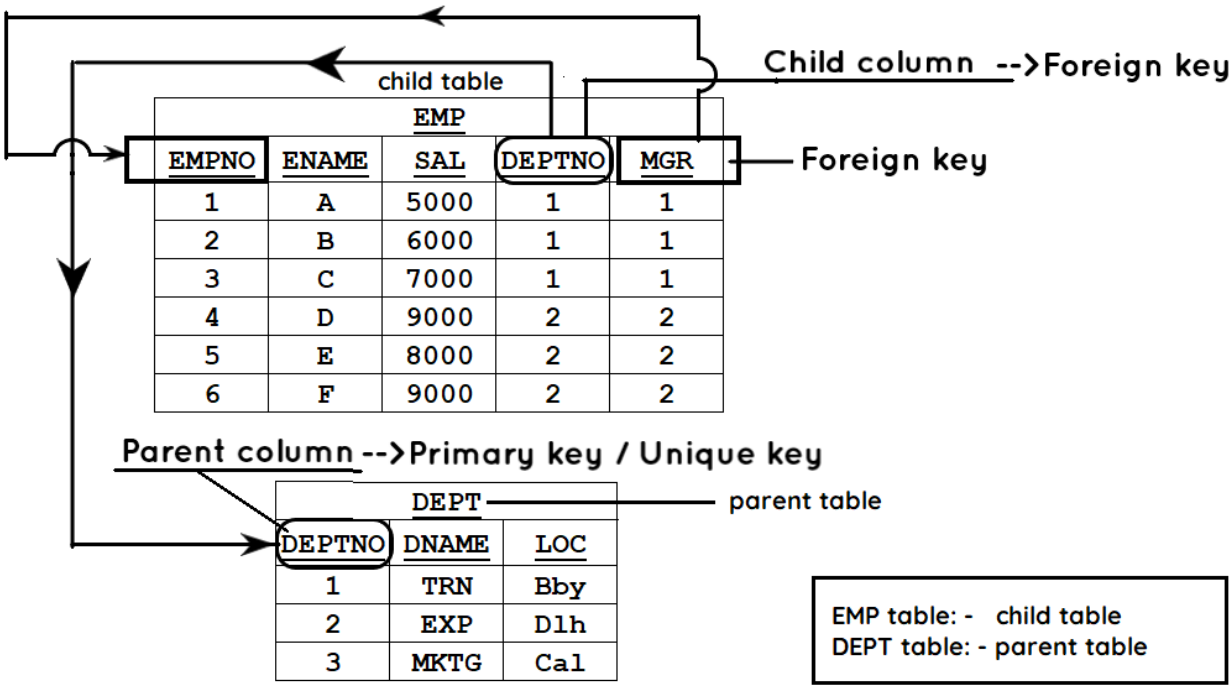
- Column or set of columns that references a column or set of columns of some table
- Foreign key constraint is specified on the child column (not the parent column)
- Parent column has to be PRIMARY KEY or UNIQUE (this is a pre-requisite for foreign key)
- Foreign key will allow duplicate values (unless specified otherwise)
- Foreign key will allow null values (unless specified otherwise)
- Foreign key may reference a column of the same table also (Known as self-referencing)
- The FOREIGN KEY constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

EMP				
EMPNO	**ENAME**	**SAL**	**DEPTNO**	**MGR**
1	A	5000	1	1
2	B	6000	1	1
3	C	7000	1	1
4	D	9000	2	2
5	E	8000	2	2
6	F	9000	2	2

DEPT		
DEPTNO	**DNAME**	**LOC**
1	TRN	Bby
2	EXP	Dlh
3	MKTG	Cal



- The table with the foreign key is called the **child table**, and
- The table with the primary key is called the **referenced or parent table**.
- **FOREIGN KEY** creates a parent-child type of relationship where the table with the FOREIGN KEY in the child table refers to the primary or unique key column in the parent table.



Creating 'dept' table as a parent table: -
(The table with the primary key is called the **referenced or parent table)**

```
create table dept
(
deptno int primary key,
dname varchar(15),
loc varchar(10)
);
```

Creating 'emp' table as a child table: -
(The table with the foreign key is called the **child table)**

```
create table emp
(
empno char(4) primary key,
ename varchar(25),
sal float,
deptno int,
mgr char(4),
constraint fk_emp_deptno foreign key(deptno)
references dept(deptno),
constraint fk_emp_mgr foreign key(mgr)
references emp(empno)
);
```

```
constraint fk_emp_deptno  → optional
constraint fk_emp_mgr → optional
```

After creating a table, if you want to find out which constraints you have created: -

```
select * from information_schema.table_constraints;
```

After creating a table, if you want to find out what constraints are present in *'juhu'* schema: -

```
select * from information_schema.table_constraints
where table_schema = 'juhu';
```

After creating a table, if you want to find out which column is having foreign key constraint: -

```
select * from information_schema.key_column_usage
where table_name = 'emp';
```

You can drop the constraint: -

```
alter table emp drop constraint fk_emp_deptno;
```

You can delete the parent row provided child rows don't exist: -

```
delete from dept where deptno = 3;
```

You cannot delete the master/parent row when matching detail/child rows exist: -

```
delete from dept where deptno = 2;
```

Solution to delete the master/parent row when matching detail/child rows exist: -

Step 1: - First delete child row

```
delete from emp where deptno = 2;
```

Step 2: - Secondly delete parent row

```
delete from dept where deptno = 2;
```

ON DELETE CASCADE:

ON DELETE CASCADE constraint is used in MySQL to delete the matching rows from the child table automatically, when the rows from the parent table are deleted.
It is a kind of referential action related to the foreign key.

```
create table emp
(
empno char(4) primary key,
ename varchar(25),
sal float,
deptno int,
mgr char(4),
constraint fk_emp_deptno foreign key(deptno)
references dept(deptno) on delete cascade,
constraint fk_emp_mgr foreign key(mgr)
references emp(empno)
);
```

on delete cascade:
If you delete the parent row, then MySQL will automatically delete the child rows also

To preserve the child rows: -

Step 1: -

```
update emp set deptno = null where deptno = 2;
```

Step 2: - Secondly delete parent row

```
delete from dept where deptno = 2;
```

You can update the parent column provided child rows don't exist

```
update dept set deptno = 4 where deptno = 3;
```

ON UPDATE CASCADE

ON UPDATE CASCADE updates the corresponding(matching) rows in the child table when the rows in the parent table are updated.

```
create table emp
(
empno char(4) primary key,
ename varchar(25),
sal float,
deptno int,
mgr char(4),
constraint fk_emp_deptno foreign key(deptno)
references dept(deptno) on delete cascade on update cascade,
constraint fk_emp_mgr foreign key(mgr)
references emp(empno)
);
```

on update cascade

If you update the parent column, then MySQL will automatically update the child rows also

To disable the foreign key constraint: -

For current connection: -

```
set foreign_key_checks = 0;
/* data manipulation */
set foreign_key_checks = 1;
```

For all connection: -

```
set global foreign_key_checks = 0;
/* data manipulation */
set global foreign_key_checks = 1;
```

CHECK constraint

- The CHECK constraint is used to limit the value range that can be placed in a column.
- If you define a CHECK constraint on a column it will allow only certain values for this column.
- If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.
- Used for validation (used for checking purposes) e.g., sal < 10000, age < 25, etc.
- Used along with Relational operators, Logical operators, Arithmetic operators, Special operators (e.g., like, in, between etc.), can call single-row functions e.g., upper, lower, etc.

```
create table emp
(
empno int,
ename varchar(25),
sal float check(sal > 5000 and sal < 3000000),
deptno int,
status char(1),
comm float,
mob_no char(15)
);
```

```
create table emp
(
empno int auto_increment primary key,
ename varchar(25) check(ename = upper(ename)),    /*column level constraint*/
sal float default 7000
check(sal between 5001 and 2999999),
deptno int,
status char(1) default 'T'
check(status in('T', 'P', 'R')),                  /*column level constraint*/
comm float not null,
mob_no char(15) unique,
check (sal+comm < 5000000),                       /*table level constraint*/
constraint fk_emp_deptno foreign key(deptno) references dept(deptno)
);
```

DEFAULT

- DEFAULT is not a constraint
- DEFAULT is a clause that you can use with CREATE TABLE
- The DEFAULT constraint is used to set a default value for a column.
- If you insert some value, then it will take that value; if nothing is entered, then it will take default value

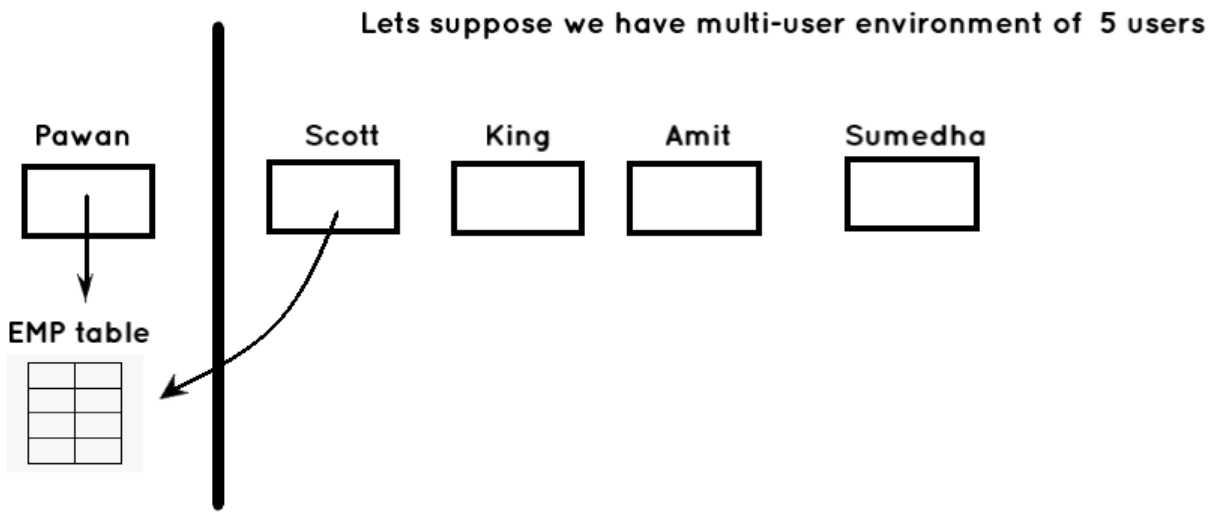
To make use of DEFAULT value and AUTO_INCREMENT, use the following INSERT statement: -

```
insert into emp(ename,deptno,comm,mob_no)
values(.....);
```

MySQL – SQL – PRIVILEGES

- In MySQL, privileges are rights or permissions granted to users that define what actions they can perform on the database objects.
- These privileges can be granted at different levels, such as the global level, database level, table level, and even column or procedure level.

GRANT and REVOKE (DCL)



GRANT

Syntax:

GRANT privileges_names ON object TO user;

- Here scott can only select from emp he cannot insert, update, delete

```
pawan_mysql> grant select on emp to scott;
```

- If scott is trusted user and you grant him permission to INSERT, UPDATE, DELETE; then he can insert, update, delete

```
pawan_mysql> grant insert on emp to scott;
pawan_mysql> grant insert on emp to scott;
pawan_mysql> grant update on emp to scott;
pawan_mysql> grant delete on emp to scott;
```

```
pawan_mysql> grant select, insert on emp to scott;
```

- If you want to grant permission to multiple users then,

```
pawan_mysql> grant select, insert on emp to scott, king, amit;
```

- To grant all permission to INSERT, UPDATE, DELETE;

```
pawan_mysql> grant all on emp to scott;
```

- To grant permission to all users

```
pawan_mysql> grant select on emp to public;
```

REVOKE

The Revoke statement is used to revoke some or all of the privileges which have been granted to a user in the past.

Syntax:

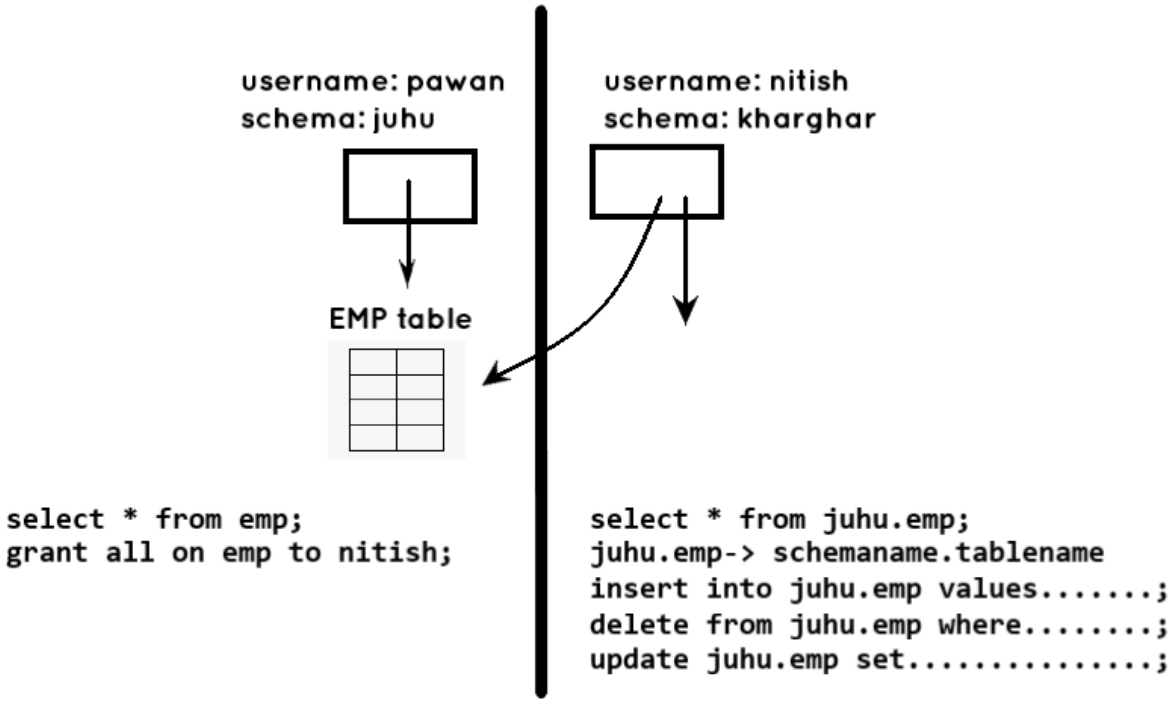
REVOKE privileges ON object FROM user;

```
pawan_mysql> revoke select on emp to scott;
```

- To see the permission received and granted

```
select * from information_schema.table_privileges;
```

NOTE: - SCHEMA IS SYNONYM FOR DATABASE



System tables

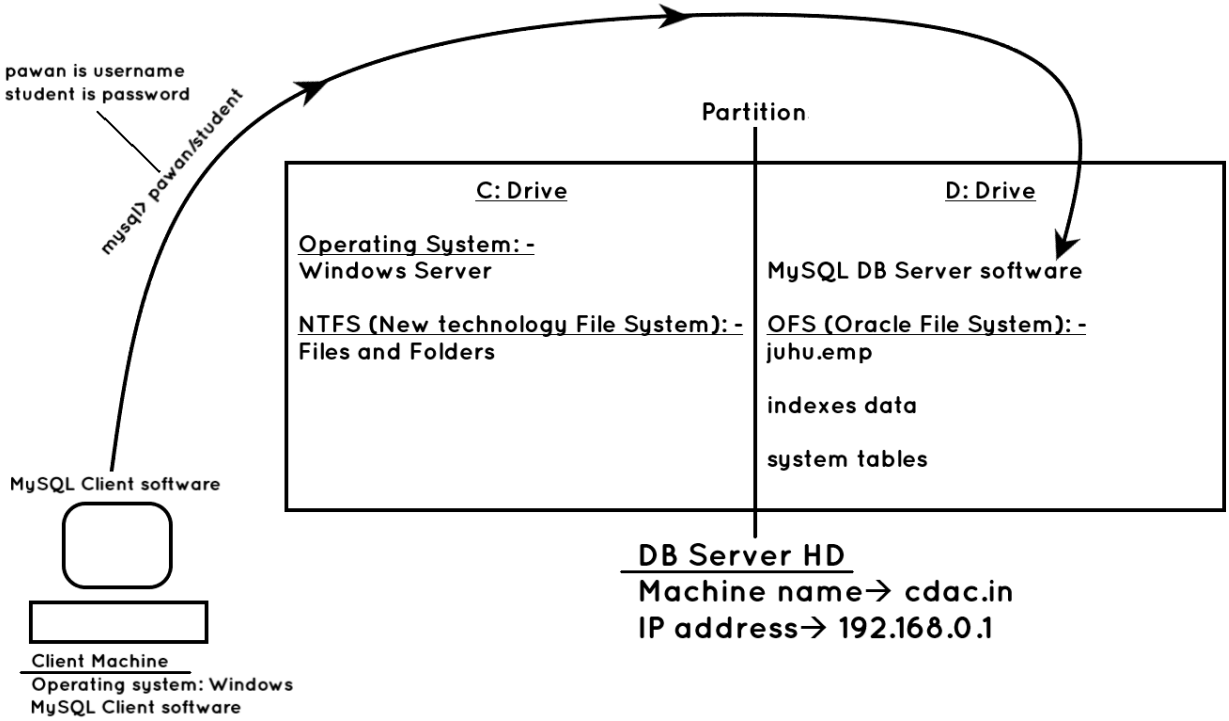
- Total of 79 System tables in MySQL
- Set of System tables are known as DATA DICTIONARY
- Also known as Database Catalog
- Automatically created when you install MySQL
- Store complete information about the database
  e.g., information of users, tables, columns, databases, indexes, constraints, privileges etc.
- All System tables are READ_ONLY
- You can only SELECT from the System tables; you cannot insert, update, or delete from the System tables
- DDL for user is DML for System tables
- E.g., statistics (for indexes), table_constraints, key_column_usage, table_privileges, etc.
- System tables are stored in information_schema

```
use information_schema;
show tables;
```

Data is of 2 types: -

1. User data
    - User created
    - User tables, user indexes
2. System data (also known as Meta data) (data about data)
    - MySQL created
    - Data that is stored in System tables
    - E.g., statistics (for indexes), table_constraints, key_column_usage, table_privileges, etc.

Architecture for MySQL



MySQL – STORED OBJECTS

- Objects that are stored in the databases
- E.g., CREATE.....tables, indexes
- Anything that you do with create command is a stored object

VIEWS

- Present in all RDBMS, and some of the DBMS also
- Handle to a table
- Views stores the address of table
- View is a HD pointer (stores the address of table) (HD pointer is known as a LOCATOR)
- Used for indirect access to the table
- Used to restrict the access of users
- Used for Security purposes

Username: pawan
Schema: juhu

EMP			
EMPNO	ENAME	SAL	DEPTNO
1	A	5000	1
2	B	6000	1
3	C	7000	1
4	D	9000	2
5	E	8000	2

Username: nitish
Schema: kharghar

Command to create view: -

```
mysql> create view viewname.....;
```

(Pawan is typing command)

```
pawan_mysql> create view v1
        as
        select empno, ename from emp;
        View Created
```

```
pawan_mysql> grant select on v1 to nitish;
```

```
nitish_mysql> select * from juhu.emp;      ← ERROR
```

```
nitish_mysql> select * from juhu.v1;
```

Schema_name = juhu
view_name = v1

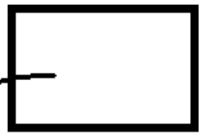
empno	ename
1	A
2	B
3	C
4	D
5	E

Username: pawan
Schema: juhu

EMP			
EMPNO	ENAME	SAL	DEPTNO
1	A	5000	1
2	B	6000	1
3	C	7000	1
4	D	9000	2
5	E	8000	2

V1 = select empno, ename from emp

Username: nitish
Schema: kharghar



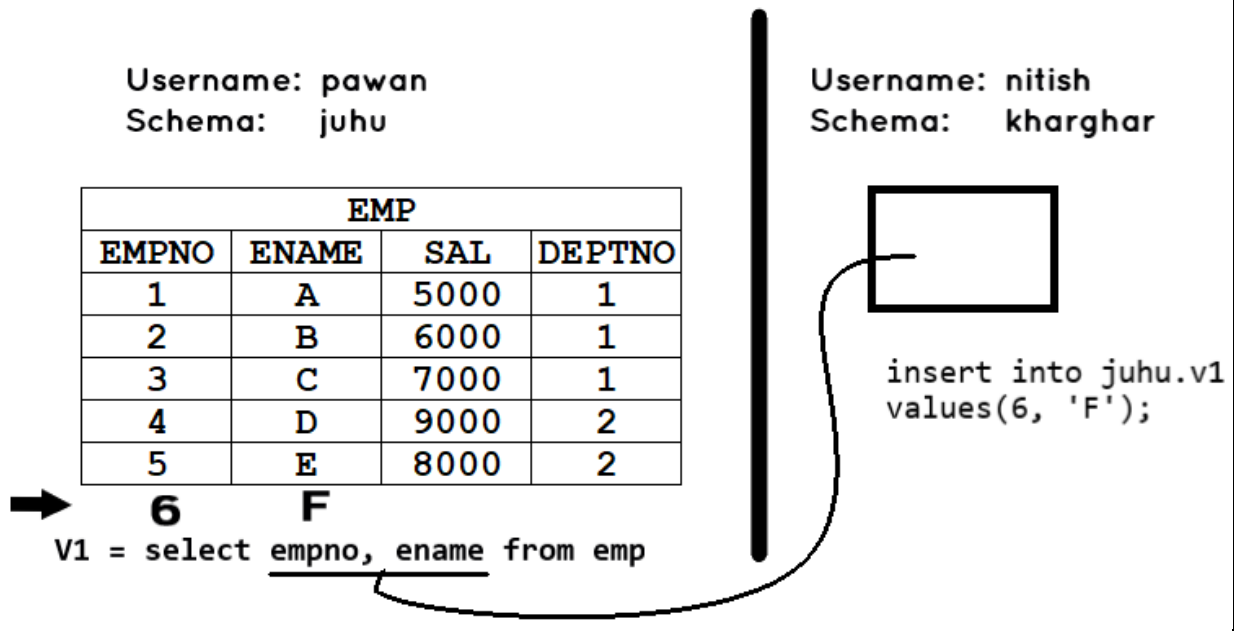
empno	ename
1	A
2	B
3	C
4	D
5	E

- Used to restrict column access
- View is a form of encapsulation (data hiding)
- View name and table name cannot be the same
- VIEW DOES NOT CONTAIN DATA
- Only the definition is stored, data is not stored
- View is a stored query (stored in the DB server HD)
- SELECT statement on which the view is based is stored in the database in the COMPILED FORMAT
- View is an executable format of SELECT statement hence the execution will be very fast
- Hiding the source code from end user

```
pawan_mysql> create view v1
        as
        select empno, ename from emp;
        View Created
```

```
pawan_mysql> grant select, insert on v1 to nitish;
```

```
nitish_mysql> insert into juhu.v1 values(6, 'F');
```



- DML operations can be performed on a view
- DML operations done on a view will affect the base table
- Constraint specified on the table will always be enforced
  Even if you insert via the view
- View can be used for Select, Insert, Update, and Delete purposes
- Entire application is built on Views

```
create view v1
as
select empno, ename from emp;
```

View Created

To DROP the VIEW: -

```
drop view v1;
```

To alter the SELECT statement on which the VIEW is based: -

```
drop view v1;

create view v1
as
select ename, sal from emp;
```

```
create view v2
as
select * from emp where deptno = 1;

select * from v2;
```

EMPNO	ENAME	SAL	DEPTNO
1	A	5000	1
2	B	6000	1
3	C	7000	1

```
insert into v2 values(6, 'F', 6000, 2);        ← IT WILL ALLOW
```

<u>WITH CHECK OPTION :</u>

- This WITH CHECK OPTION prevents you from updating or inserting rows that are not visible through the view

To restrict the user, from inserting or updating rows other than rows with deptno = 1 i.e., he can insert or update only those rows whose deptno = 1

```
create view v2
as
select * from emp where deptno = 1 with check option;

insert into v2 values(6, 'F', 6000, 2);        ← ERROR
```

- View WITH CHECK OPTION is similar to check constraint
- Used to enforce different checks for different users

<u>with check option</u> is more powerfull than <u>check constraint</u>
Suppose you have 3 users, user A, user B, user C
CASE 1: If you have check constraint on sal(salary) column of sal is less than 10000 then that constraint is applicable for all users i.e, no one can insert salary above 10000)
CASE 2: If A is inserting, he can insert salary upto 10000, If B is inserting, he can insert salary upto 20000, whereas If C is inserting he can insert salary upto 30000, i.e.,
different checks for ddifferent users then what would be solution ?
Solution: - create 3 Views,
create view v1 with sal<10000 with check option,
create view v2 with sal<20000 with check option,
create view v3 with sal<30000 with check option
and then
for user A, grant insert permission for v1,
for user B, grant insert permission for v1,
for user C, grant insert permission for v1,

```
create or replace view v1
as
select ename, sal from emp;
```

Describe view

```
desc v1;
```

```
create or replace view v1
as
select ename, sal*12 as annual from emp;

select * from v1;
```

- view based on computed column, expression, function, ORDER BY clause, GROUP BY clause, and distinct
- you can only SELECT from this view
- DML operations are not allowed
- this is common for all RDBMS

View based on JOIN

```
create or replace view v1
as
select dname, ename from emp, dept
where dept.deptno = emp.deptno;

select * from v1;
```

- view based on Join
- you can only SELECT from this view
- DML operations are not allowed

```
desc v1;
show tables;
```

will show tables and views but it won't tell us which is a table and which is a view

To find out which is a table and which is a view: -

```
show full tables;
```

To see the SELECT statement on which the view is based: -

```
show create view v1;
```

View based on view is allowed

Uses of VIEWS:

- Used for Security purposes
- Used to restrict column access and row access
- View is an executable format of SELECT statement hence the execution will be very fast
- Hiding the source code from end user
- View WITH CHECK OPTION is similar to check constraint
- Used to enforce different checks for different users
- View based on view is allowed
- to exceed the limits of SQL
  e.g.,
- UNION of > 255 SELECT statements
- Sub-queries > 255 levels
- Functions within function > 255 levels
- To simplify the writing of complex, SELECT statements
  e.g.,
  Join of 20 tables

MySQL – PL (Programming Language)

- Product of MySQL
- MySQL Programming Language
- Used for database programming
  (e.g., HRA_CALC, TAX_CALC, ATTENDANCE_CALC, etc.)
- Used for Server-side data processing
- MySQL-PL program can be called through MySQL Command Line Client, MySQL Workbench, Java, MS .Net, etc.; can be called through any front-end software
- Few 4 GL features
- MySQL-PL program is referred to a MySQL-PL block
- Block level language
- Benefits of Block level language: -
  a. Modularity
  b. Control scope of variables (form of Encapsulation)
     (Form of data hiding)
  c. Efficient Exception handling (error management)
- Screen input and screen output not allowed inside the program
  (scanf, printf, etc. not available)
- Used ONLY for processing
- You can use SELECT statement inside the block but it is not recommended; because you will encounter a problem when you call your program through front-end software
- You can use SELECT statement in the program if it is a part of sub-query, because then there is no output coming on the screen e.g.,

```
delete from emp where deptno = (select deptno from emp where ename = 'Thomas');
```

- SQL commands that are allowed within MYSQL-PL are: -
  DDL, DML, DQL, DTL/TCL
- DCL commands are not allowed within program

MySQL-PL programs consists of following: -

```
Begin
    Insert into emp values(1,'a','B');   } MySQL PL Block
End;
```

Block within Block: -

```
Begin                    ← Main Block
    ……….
    ……….
        Begin            ← Sub Block
            ……….
            ……….
        End;
    ……….
    ……….
End;
```

Execution is from top to bottom

Every RDBMS has its own native programming language: -
Oracle → PL/SQL → Procedural Language SQL
MS SQL Server → T-SQL → Transact SQL
MySQL → MySQL-PL → MySQL Programming Language

Temporary Tables

The **Temporary Tables** are the tables that are created in a database to store data temporarily. These tables will be automatically deleted once the current client session is terminated or ends (permanent storage is not allocated in database). In addition to that, these tables can be deleted explicitly if the users decide to drop them manually.
You can perform various SQL operations on temporary tables, just like you would with permanent tables, including CREATE, UPDATE, DELETE, INSERT, JOIN, etc.
Creating a temporary table in MySQL is very similar to creating a regular database table. But, instead of using CREATE TABLE, we use CREATE TEMPORARY TABLE statement. (The only difference is that you must specify the temporary keyword between create and table keywords)

Syntax

```
CREATE TEMPORARY TABLE table_name(
   column1 datatype,
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
   PRIMARY KEY(one or more columns )
);
```

There are three ways to create a temporary table:

1. **First Method**: First method is to create a table as you normally create table just write TEMPORARY keyword in between CREATE and TABLE

```
CREATE TEMPORARY TABLE [Table Name] (Column Name ....);
```

2. **Second Method**: Use the LIKE keyword to copy the structure of an existing table. For example:

```
CREATE TEMPORARY TABLE [New Table] LIKE [Original Table];
```

3. **Third Method**: Use the SELECT statement to copy both the data and the structure of an existing table. For example:

```
CREATE TEMPORARY TABLE [New Table] AS SELECT * FROM [Original
Table];
```

Dropping Temporary Tables in MySQL

```
DROP TEMPORARY TABLE table_name;
DROP TEMPORARY TABLE CUSTOMERS;
```

- MySQL removes the temporary table automatically when the session ends or the connection is terminated. Also, you can use the DROP TABLE statement to remove a temporary table explicitly when you are no longer using it.
- A temporary table is only available and accessible to the client that creates it. Different clients can create temporary tables with the same name without causing errors because only the client that creates the temporary table can see it. However, in the same session, two temporary tables cannot share the same name.
- A temporary table can have the same name as a regular table in a database. For example, if you create a temporary table named employees in the sample database, the existing employees table becomes inaccessible. Every query you issue against the employees table is now referring to the temporary table employees. When you drop the employees temporary table, the regular employees table is available and accessible.
- Even though a temporary table can have the same name as a regular table, it is not recommended. Because this may lead to confusion and potentially cause an unexpected data loss. For example, if the connection to the database server is lost and you reconnect to the server automatically, you cannot differentiate between the temporary table and the regular one.
- Then, you may issue a DROP TABLE statement to remove the permanent table instead of the temporary table, which is not expected.
- To avoid this issue, you can use the DROP TEMPORARY TABLE statement to drop a temporary table instead of the DROP TABLE statement

```
CREATE TEMPORARY TABLE temp_sales
(
    sale_id INT,
    product_id INT,
    sale_amount DECIMAL(10, 2)
);

INSERT INTO temp_sales (sale_id,
product_id, sale_amount) VALUES
(1, 101, 150.00),
(2, 102, 200.00),
(3, 101, 250.00),
(4, 103, 300.00);
```

sale_id	product_id	sale_amount
1	101	150.00
2	102	200.00
3	101	250.00
4	103	300.00

Querying a temporary table is identical to querying a regular table.
For example, to get the total sales amount from the temporary table:

```
SELECT SUM(sale_amount) AS total_sales
FROM temp_sales;
```

total_sales
900.00

Temporary Table Whose Structure is Based on a Normal Table
First, we create two normal tables: customers and orders.

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(100),
    email VARCHAR(100),
    phone VARCHAR(20)
);

INSERT INTO customers
(customer_id, customer_name,
email, phone) VALUES
(1, 'John Doe',
'john.doe@example.com', '123-
456-7890'),
(2, 'Jane Smith',
'jane.smith@example.com', '987-
654-3210');
```

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    amount DECIMAL(10, 2),
    FOREIGN KEY (customer_id)
REFERENCES
customers(customer_id)
);

INSERT INTO orders (order_id,
customer_id, order_date, amount)
VALUES
(1, 1, '2024-06-30', 150.00),
(2, 2, '2024-06-30', 200.00),
(3, 1, '2024-07-01', 250.00);
```

Create a Temporary Table Based on orders Structure
We will create a temporary table named "`temp_orders`" based on the structure of the orders table.

```
CREATE TEMPORARY TABLE temp_orders LIKE orders;
```

- This command creates the "`temp_orders`" table with the same structure as the `orders` table, but without any data.

Create a Temporary Table to copy both the data and the structure of customers existing table.

```
CREATE TEMPORARY TABLE temp_customers AS SELECT * FROM customers;
```

- This command creates the "`temp_customers`" table with the same structure as the `customers` table, along with its data.

DAY 11

- MySQL PL program is written in the form of Stored procedure

MySQL – STORED OBJECTS

- Objects that are stored in the database
  e.g., CREATE..... tables, indexes, views
- Anything that you do with CREATE command is stored object

STORED PROCEDURES

A procedure (often called a stored procedure) is a collection of pre-compiled SQL statements stored inside the database. In short, A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

A procedure is a subroutine (like a subprogram) in a regular scripting language, stored in a database. In the case of MySQL, procedures are written in MySQL and stored in the MySQL database/server. A MySQL procedure has a name, a parameter list, and SQL statement(s).

- Routine (set of commands) that has to be called explicitly
  E.g.,
  ```
  mysql> call hra_calc();
  ```
- Global procedures
- Can be called from MySQL Command Line Client, MySQL Workbench, Oracle Forms, Oracle Reports, Oracle Menus, Oracle Graphics, Oracle Apex (Application Express), Java, MS .Net, etc.
- Can be called through any front-end software
- Stored in the database in the COMPILED FORMAT
- Hence the execution will be very fast
- Hiding source code from end user
- Execution takes place in Server RAM
- Procedure can have local variables
- Within a procedure all MySQL-PL statements allowed e.g., IF statement, loops, cursors, etc.
- One procedure can call another procedure
  (Ensure to create called procedure first and calling procedure afterwards)
- Procedure can call itself (known as Recursion)
- You can pass parameter to a procedure (to make it flexible); the same procedure can be run for different values
- OVERLOADING OF STORED PROCEDURES IS NOT ALLOWED because it is stored object
- You cannot create 2 or more procedures with the same name even if the NUMBER of parameters passed is different or the DATATYPE of parameters passed is different

Need/ Significance of Delimiter

- When you want to execute multiple SQL statements, you use the semicolon (;) to separate two statements, as shown in the following example:

```
SELECT * FROM products;
SELECT * FROM customers;
```

- A MySQL client program, such as MySQL Workbench or the mysql program, uses the default delimiter (;) to separate statements and execute each separately.
- However, a stored procedure consists of multiple statements separated by a semicolon (;).
- If you use a MySQL client program to define a complex statements or stored procedure that contains semicolons, the MySQL client program will not treat the entire stored procedure as a single statement; instead, it will recognize it as multiple statements. Semicolons within the statement can cause issues with execution. Therefore, it is necessary to temporarily redefine the delimiter so that you can pass the entire stored procedure to the server as a single statement.

To redefine the default delimiter, you use the DELIMITER command as follows:

```
DELIMITER delimiter_character
```

- The `delimiter_character` may consist of a single character or multiple characters, such as // or $$. However, you should avoid using the backslash (\) because it's the *escape character* in MySQL.

MySQL uses backslash (\) as the escape character, which allows you to include special characters within strings without triggering syntax errors. For example, you can use the escape character to include a single quote in a string like this: SELECT 'It\'s a sunny day';

The following example illustrates how to change the current delimiter to //

```
DELIMITER //
```

After you change the delimiter, you can use the new delimiter to end a statement, as follows:

```
DELIMITER //

SELECT * FROM customers //

SELECT * FROM products //
```

To revert to the default delimiter, which is a semicolon (;), you use the following statement:

```
DELIMITER ;
```

Program:

Creating table to store output of MySQL-PL program: -

```
create table tempp
(
fir int,
sec char(15)
);
```

TEMPP	
FIR	SEC

- Table name and procedure name cannot be same (maximum 30 characters allowed)
- **begin** indicates start of procedure and **end** indicates exit of procedure

```
create procedure abc()
begin
    insert into tempp values(1, 'Hello');
end;

Procedure created.
```

ERROR

- Read, Compile, Plan and Store it in the DB in the COMPILED FORMAT
- **;** is known as delimiter (indicates end of command)

- MySQL environment is interpreter based, when the system is reading this code it will compile it, recognize it as multiple statements and not as a single unit (because there are multiple semicolons in code) due to it will encounter ERROR
- To handle this, MySQL provides the `DELIMITER` command, which allows you to temporarily change the statement delimiter so that semicolons can be used within the procedure body without causing issues. Here's how it works:

Creating a Stored Procedure with Multiple Statements

1. **Change the Delimiter:**

Use the `DELIMITER` command to change the statement delimiter from the default semicolon (`;`) to another string, such as `$$` or `//`.

2. **Define the Procedure:**

Write the `CREATE PROCEDURE` statement, using semicolons to separate the SQL statements within the procedure body.

3. **Reset the Delimiter:**

After defining the procedure, change the delimiter back to the default semicolon (`;`).

```
delimiter //

create procedure abc()
begin
    insert into tempp values(1, 'Hello');
end; //

delimiter ;

Procedure created.
```

Change the Delimiter

Define the Procedure

Reset the Delimiter

To call the procedure: -

```
mysql> call abc();
                OR
mysql> call abc;
```

NOTE: -

- Stored procedures that take no arguments can be invoked without parentheses. That is, CALL abc() and CALL abc are equivalent.

In short, While calling a stored procedure that doesn't accepts any arguments, we can omit the parenthesis

To call the procedure: -

```
mysql> select * from tempp;
```

```
+------+-------+
| fir  | sec   |
+------+-------+
|    1 | Hello |
+------+-------+
```

To drop the procedure: -

```
drop procedure abc;
```

NOTE: -

If you try to drop a procedure that doesn't exist, an error will be generated as shown below –

DROP PROCEDURE abc;

(MySQL issued the following error:

```
Error Code: 1305. PROCEDURE classicmodels.abc does not exist)
```

If you use the **IF EXISTS** clause along with the DROP PROCEDURE statement as shown below, the specified procedure will be dropped and if a procedure with the given name, doesn't exist the query will be ignored.

Using MySQL DROP PROCEDURE with the IF EXISTS option example

```
DROP PROCEDURE IF EXISTS abc;
```

```
delimiter //
create procedure abc()
begin
    insert into tempp values(1, 'Hello');
end; //

delimiter ;

call abc();
```

- While calling a stored procedure that doesn't accepts any arguments, we can omit the parenthesis

```
call abc;

select * from tempp;
```

```
+------+-------+
| fir  | sec   |
+------+-------+
|    1 | Hello |
+------+-------+
```

Program 1 -

```
delimiter //
create procedure abc1()
begin
    declare x int;
    set x = 10;
    insert into tempp values(x, 'Hello');
end; //
delimiter ;

call abc1();

select * from tempp;
```

Scope of x variable is limited to this block (Local variables)

```
+------+-------+
| fir  | sec   |
+------+-------+
|   10 | Hello |
+------+-------+
```

Here,
`set x = 10;` is an assignment operator

- Variables are created in Server RAM
- Even though you call procedure from your computer, Execution takes place in Server RAM
- In MySQL-PL, when you declare a variable, if you don't initialize it, then it will store a null value
- Variables must be declared at the top of the BEGIN...END block, immediately following the BEGIN statement using the DECLARE statement.
- You can declare a variable and assign a value to it simultaneously

Program 2

```
delimiter //
create procedure abc2()
begin
    declare x char(15) default 'CDAC';
    insert into tempp values(1, x);
end; //
delimiter ;

call abc2();

select * from tempp;
+------+------+
| fir  | sec  |
+------+------+
|    1 | CDAC |
+------+------+
```

Note: - to assign values to `CHAR`, `VARCHAR`, `DATE`, `TIME`, and `DATETIME` variables, you should use single quotes (') to enclose the values e.g., 'CDAC'

Program 3 (HRA Calculation / variables with default values)

```
delimiter //
create procedure abc3()
begin
    declare x char(15) default 'KING';
    declare y float default 3000;
    declare z float default 0.4;
    declare hra float;
    set hra = y*z;
    insert into tempp values(y, x);
    insert into tempp values(hra, 'HRA');
end; //
delimiter ;

call abc3();

select * from tempp;
+------+------+
| fir  | sec  |
+------+------+
| 3000 | KING |
| 1200 | HRA  |
+------+------+
```

Parameterized procedure (Stored Procedure with an Input Parameter)

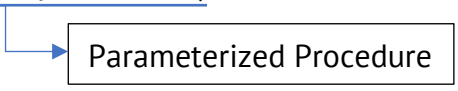
A parameterized procedure is a type of stored procedure that can accept input parameters. These parameters can be used to customize the behaviour of the procedure and perform operations based on the input values provided.

Program 4 (HRA Calculation with parameters)

```
delimiter //
create procedure abc4(x char(15), y float, z float)
begin
    declare hra float;
    set hra = y*z;
    insert into tempp values(y, x);
    insert into tempp values(hra, 'HRA');
end; //
delimiter ;

call abc4('KING', 3000, 0.4);
call abc4('SCOTT', 2500, 0.3);

select * from tempp;
+------+-------+
| fir  | sec   |
+------+-------+
| 3000 | KING  |
| 1200 | HRA   |
| 2500 | SCOTT |
|  750 | HRA   |
+------+-------+
```



COMMENTS

- -- Single line comment
- /* Multi-Line
  Comment */

Note: -

- You must specify a comment, minimum every 2 statements
- Min 33% of code is comments
- Normal discipline of writing code is in beginning
  (Specify your name, address, email-id, contact details)

Program 5 (HRA Calculation / selecting data from tables(emp))

```
create table tempp
(
fir int,
sec char(15)
);
```

TEMPP

FIR	SEC

```
create table emp(
ENAME varchar(20),
SAL int,
JOB varchar(20),
DEPTNO int);

insert into emp(ENAME, SAL, JOB, DEPTNO)
values
('SCOTT',3000,'CLERK',10),
('KING',5000,'MANAGER',20);
```

EMP

ENAME	SAL	JOB	DEPTNO
SCOTT	3000	CLERK	10
KING	5000	MANAGER	20

```
delimiter //
create procedure abc5()
begin
    declare x int;
    select sal into x from emp
    where ename = 'KING';
    /* processing, e.g., set hra = x*0.4, etc. */
    insert into tempp values(x, 'KING');
end; //
delimiter ;

call abc5;

select * from tempp;
```

EMP

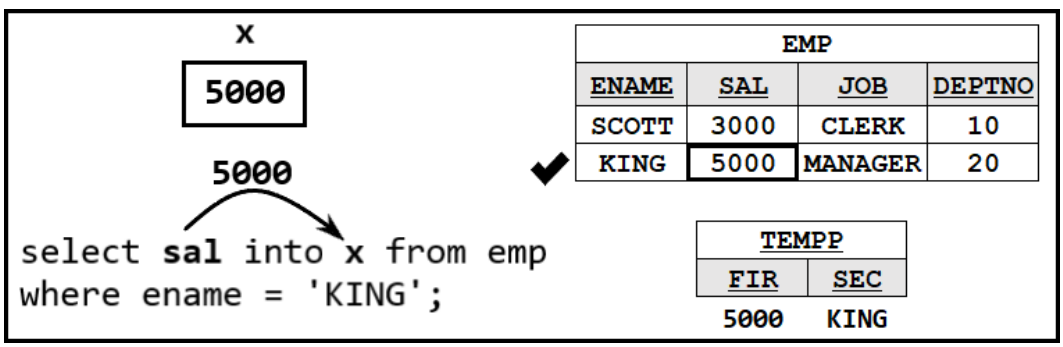
ENAME	SAL	JOB	DEPTNO
SCOTT	3000	CLERK	10
KING	5000	MANAGER	20

TEMP

FIR	SEC

```
+------+------+
| fir  | sec  |
+------+------+
| 5000 | KING |
+------+------+
```

- x variable and sal column datatype must be same



Program 6 (HRA Calculation/ selecting data from tables(emp), taking input from user)

```
delimiter //
create procedure abc6(y char (15))
begin
    declare x int;
    select sal into x from emp
    where ename = y;
    /* processing, e.g., set hra = x*0.4, etc. */
    insert into tempp values(x, y);
end; //
delimiter ;

call abc6('KING');
call abc6('SCOTT');

select * from tempp;
```

```
+------+-------+
| fir  | sec   |
+------+-------+
| 5000 | KING  |
| 3000 | SCOTT |
+------+-------+
```

Program 7

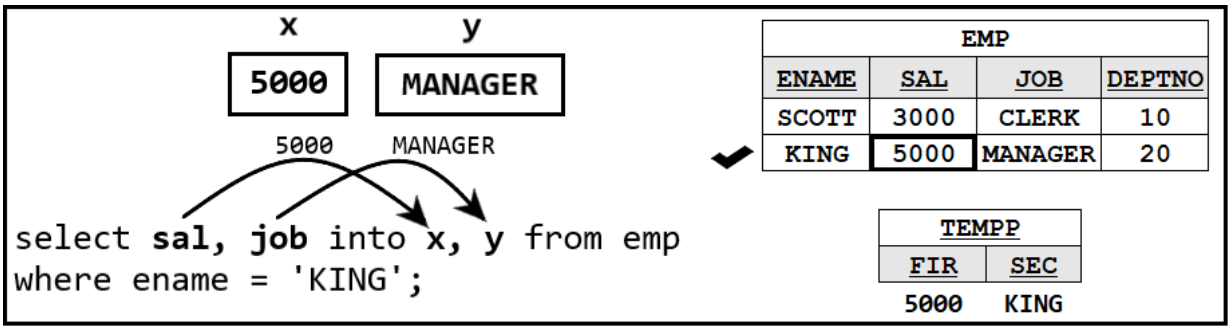
```
delimiter //
create procedure abc7()
begin
    declare x int;
    declare y char (15);
    select sal, job into x, y from emp
    where ename = 'KING';
    /* processing, e.g., set hra = x*0.4, set y = lower(y) etc. */
    insert into tempp values(x, y);
end; //
delimiter ;

call abc7;

select * from tempp;
```

```
+------+---------+
| fir  | sec     |
+------+---------+
| 5000 | MANAGER |
+------+---------+
```

- x variable should have same datatype as sal column
- y variable should have same datatype as job column



To drop the procedure: -

```
drop procedure abc;
```

To see which all procedures are created: -

```
show procedure status;
```
← shows all procedures in all databases/schemas

To see all procedures present in specific database: -

```
show procedure status where db = 'juhu';
```

- shows all procedures in juhu database/schema

```
show procedure status where name like 'a%';
```

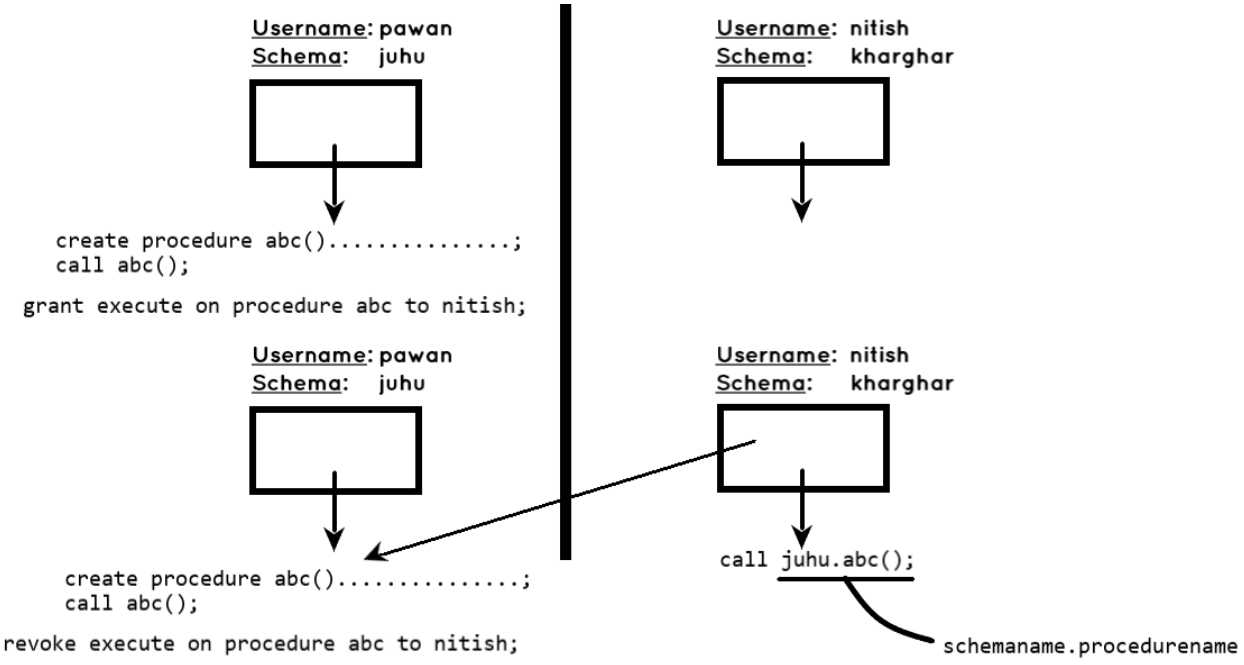
- shows all procedures which are starting with letter **'a'** in databases/schemas

To view source code of stored procedure: -

```
SHOW CREATE PROCEDURE procedure_name
```

```
show create procedure abc;
```

To share procedure with others: -




```
grant execute on procedure abc to nitish;
```

To revoke procedure with others: -

```
revoke execute on procedure abc to nitish;
```

Summary:-

A procedure (often called a stored procedure) is a collection of pre-compiled SQL statements stored inside the database. In short, A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

The syntax to create a MySQL Stored procedure is the following:

```
DELIMITER //
CREATE PROCEDURE procedure_name(parameter_1, parameter_2, . . ., parameter_n)
BEGIN
    instruction_1;
    instruction_2;
    . . .
    instruction_n;
END //
DELIMITER ;
```

In the syntax:

- The name of the procedure must be specified after the **Create Procedure** keyword
- After the name of the procedure, the list of parameters must be specified in the parenthesis. The parameter list must be comma-separated
- The SQL Queries and code must be written between **BEGIN** and **END** keywords

To execute the store procedure, you can use the CALL keyword. Below is syntax:

```
CALL [Procedure Name] ([Parameters]....);
```

- While calling a stored procedure that doesn't accepts any arguments, we can omit the parenthesis

Up till now concepts covered

```
MySQL-PL block, Begin .... End;
Concept of TEMPP output table,
Declare variables, Intialize variables, Default value for variable
Pass parameters
Processing, e.g. hra_calc
Select col1, col2, ...., coln into var1, var2, ...., varn from .... where .....
```

Decision making using IF statement

```
create table tempp
(
fir int,
sec char(15)
);
```

TEMPP	
FIR	SEC

```
create table emp(
ENAME varchar(20),
SAL int,
JOB varchar(20),
DEPTNO int);

insert into emp(ENAME, SAL, JOB, DEPTNO)
values
('SCOTT',3000,'CLERK',10),
('ADAM',5000,'MANAGER',20),
('JONES',8000,'CLERK',30),
('QUEEN',7000,'MANAGER',20),
('KING',9000,'MANAGER',20);
```

EMP			
ENAME	SAL	JOB	DEPTNO
SCOTT	3000	CLERK	10
ADAM	5000	MANAGER	20
JONES	8000	CLERK	30
QUEEN	7000	MANAGER	20
KING	9000	MANAGER	20

```
delete from tempp; drop procedure abc1;
```

Program 1

```
delimiter //
create procedure abc1()
begin
    declare x int;
    select sal into x from emp where ename = "KING";
    if x > 5000 then
        insert into tempp values(x, 'High sal');
    end if;
end; //
delimiter ;

call abc1;

select * from tempp;
```

```
+------+----------+
| fir  | sec      |
+------+----------+
| 9000 | High sal |
+------+----------+
```

x

9000

EMP			
ENAME	SAL	JOB	DEPTNO
SCOTT	3000	CLERK	10
ADAM	5000	MANAGER	20
JONES	8000	CLERK	30
QUEEN	7000	MANAGER	20
KING	9000	MANAGER	20

```
+------+----------+
| fir  | sec      |
+------+----------+
| 9000 | High sal |
+------+----------+
```

```
if condition_is_satisfied then
    ……;
    ……;
end if;
```

Program 2

```
delete from tempp; drop procedure abc2;
```

```
delimiter //
create procedure abc2()
begin
    declare x int;
    select sal into x from emp where ename = "KING";
    if x > 5000 then
        insert into tempp values(x, 'High sal');
    else
        insert into tempp values(x, 'Low sal');
    end if;
end; //
delimiter ;

call abc2;

select * from tempp;
```

```
+------+----------+
| fir  | sec      |
+------+----------+
| 9000 | High sal |
+------+----------+
```

- Writing else statement is optional

Program 3

```
delete from tempp; drop procedure abc3;
```

```
delimiter //
create procedure abc3()
begin
    declare x int;
    select sal into x from emp where ename = "KING";
    if x > 5000 then
        insert into tempp values(x, 'High sal');
    else
        if x < 5000 then
            insert into tempp values(x, 'Low sal');
        else
            insert into tempp values(x, 'Medium sal');
        end if;
    end if;
end; //
delimiter ;

call abc3;

select * from tempp;
```

- Writing else statement is optional

Program 4: ELSEIF

```
delete from tempp; drop procedure abc4;
```

```
delimiter //
create procedure abc4()
begin
    declare x int;
    select sal into x from emp where ename = "KING";
    if x > 5000 then
        insert into tempp values(x, 'High sal');
    elseif x < 5000 then
        insert into tempp values(x, 'Low sal');
    else
        insert into tempp values(x, 'Medium sal');
    end if;
end; //
delimiter ;

call abc4;

select * from tempp;
```

- Writing else statement is optional

Program 5

```
delete from tempp; drop procedure abc5;
```

```
delimiter //
create procedure abc5()
begin
    declare x boolean default TRUE;
    if x then
        insert into tempp values(1, 'Mumbai');
    end if;
end; //
delimiter ;

call abc5;

select * from tempp;
```

```
+------+--------+
| fir  | sec    |
+------+--------+
|    1 | Mumbai |
+------+--------+
```

- Boolean is logical datatype (you can use true or false)

Program 6

```
delete from tempp; drop procedure abc6;
```

```
delimiter //
create procedure abc6()
begin
    declare x boolean default FALSE;
    if not x then
        insert into tempp values(1, 'Delhi');
    end if;
end; //
delimiter ;

call abc6;

select * from tempp;
```

```
+------+--------+
| fir  | sec    |
+------+--------+
|    1 | Mumbai |
| 9000 | KING   |
+------+--------+
```

CASE statement

```
drop procedure abc;
```

```
delimiter //
create procedure abc()
begin
    declare x int;
    select sal into x from emp where ename = 'KING';
    case
    when x > 5000 then
        insert into tempp values(x, 'High sal');
    when x < 5000 then
        insert into tempp values(x, 'Low sal');
    else
        insert into tempp values(x, 'Medium sal');
    end case;
end; //
delimiter ;

call abc;

select * from tempp;
```

```
+------+----------+
| fir  | sec      |
+------+----------+
| 9000 | High sal |
+------+----------+
```

MySQL – PL – LOOPS

- For repetitive / iterative processing

While Loop

- Check for some condition before entering the loop

Syntax

```
WHILE expression Do
    ……;
    ……;
END WHILE;
```

Program 1

```
delete from tempp; drop procedure abc1;
```

```
delimiter //
create procedure abc1()
begin
    declare x int default 1;
    while x < 10 do
        insert into tempp values(x, 'in while loop');
        set x = x + 1;
    end while;
end; //
delimiter ;

call abc1;

select * from tempp;
```

```
+------+--------------+
| fir  | sec          |
+------+--------------+
|    1 | in while loop |
|    2 | in while loop |
|    3 | in while loop |
|    4 | in while loop |
|    5 | in while loop |
|    6 | in while loop |
|    7 | in while loop |
|    8 | in while loop |
|    9 | in while loop |
+------+--------------+
```

Program 2

```
delete from tempp; drop procedure abc2;
```

```
delimiter //
create procedure abc2()
begin
    declare x int default 1;
    declare y int default 1;
    while x < 10 do
        while y < 10 do
            insert into tempp values(y, 'in y loop');
            set y = y + 1;
        end while;
        insert into tempp values(x, 'in x loop');
        set x = x + 1;
    end while;
end; //
delimiter ;
```

```
call abc2;
```

```
select * from tempp;
```

```
+------+-----------+
| fir  | sec       |
+------+-----------+
|    1 | in y loop |
|    2 | in y loop |
|    3 | in y loop |
|    4 | in y loop |
|    5 | in y loop |
|    6 | in y loop |
|    7 | in y loop |
|    8 | in y loop |
|    9 | in y loop |
|    1 | in x loop |
|    2 | in x loop |
|    3 | in x loop |
|    4 | in x loop |
|    5 | in x loop |
|    6 | in x loop |
|    7 | in x loop |
|    8 | in x loop |
|    9 | in x loop |
+------+-----------+
```

Explanation

```
delimiter //
create procedure abc2()
begin
	declare x int default 1;
	declare y int default 1;
	while x < 10 do
		while y < 10 do
			insert into tempp values(y, 'in y loop');
			set y = y + 1;
		end while;
		insert into tempp values(x, 'in x loop');
		set x = x + 1;
	end while;
end; //
delimiter ;
```

```
+------+-----------+
| fir  | sec       |
+------+-----------+
|    1 | in y loop |
|    2 | in y loop |
|    3 | in y loop |
|    4 | in y loop |
|    5 | in y loop |
|    6 | in y loop |
|    7 | in y loop |
|    8 | in y loop |
|    9 | in y loop |
|    1 | in x loop |
|    2 | in x loop |
|    3 | in x loop |
|    4 | in x loop |
|    5 | in x loop |
|    6 | in x loop |
|    7 | in x loop |
|    8 | in x loop |
|    9 | in x loop |
+------+-----------+
```

- ☒ **DECLARE x INT DEFAULT 1**: Declares a local variable x and initializes it to 1.
- **DECLARE y INT DEFAULT 1**: Declares a local variable y and initializes it to 1.

☒ **Outer WHILE Loop**

```
WHILE x < 10 DO
  ...
  SET x = x + 1;
END WHILE;
```

- **Condition**: The loop runs as long as x is less than 10.
- **SET x = x + 1**: After executing the inner loop and the outer loop's insert statement, increments x by 1.

☒ **Inner WHILE Loop**

```
WHILE y < 10 DO
  INSERT INTO tempp VALUES (y, 'in y loop');
  SET y = y + 1;
END WHILE;
```

- **Condition**: The loop runs as long as y is less than 10.
- **INSERT INTO tempp VALUES (y, 'in y loop')**: Inserts a row into the tempp table with the current value of y and the string 'in y loop'.
- **SET y = y + 1**: Increments y by 1 after each iteration of the inner loop.

☒ **After Inner Loop**

```
INSERT INTO tempp VALUES (x, 'in x loop');
SET y = 1;  -- Reset y to 1 for the next iteration of the outer loop
```

- **INSERT INTO tempp VALUES (x, 'in x loop')**: After the inner loop finishes, inserts a row into the tempp table with the current value of x and the string 'in x loop'.
- **SET y = 1**: Resets y to 1 for the next iteration of the outer loop to ensure the inner loop will run again.

☒ **Completion**

- When x reaches 10, the outer while loop condition (x < 10) is no longer satisfied, and the loop terminates.

Example Walkthrough

Initial Values

- `x = 1`
- `y = 1`

First Outer Loop Iteration (`x = 1`):

- **Inner Loop**: Executes while `y` is less than 10.
  - `y = 1`: Insert `(1, 'in y loop')`, `y` becomes 2.
  - `y = 2`: Insert `(2, 'in y loop')`, `y` becomes 3.
  - ...
  - `y = 9`: Insert `(9, 'in y loop')`, `y` becomes 10.
  - Inner loop exits because `y` is not less than 10.
- **Action**: Insert `(1, 'in x loop')`.
- **Increment**: `x` becomes 2.
- **Reset**: `y` is reset to 1.

Second Outer Loop Iteration (x = 2):

- **Inner Loop**: Executes while `y` is less than 10.
  - `y = 1`: Insert `(1, 'in y loop')`, `y` becomes 2.
  - `y = 2`: Insert `(2, 'in y loop')`, `y` becomes 3.
  - ...
  - `y = 9`: Insert `(9, 'in y loop')`, `y` becomes 10.
  - Inner loop exits because `y` is not less than 10.
- **Action**: Insert `(2, 'in x loop')`.
- **Increment**: `x` becomes 3.
- **Reset**: `y` is reset to 1.

Continuing the Pattern

The outer loop continues incrementing `x` and resetting `y` after each full iteration of the inner loop until `x` reaches 10.

Final Inserts

For each value of `x` from 1 to 9, the inner loop will insert values `(1, 'in y loop')` to `(9, 'in y loop')`, and after the inner loop completes, the outer loop will insert `(x, 'in x loop')`.

☒ The outer loop iterates x from 1 to 9.
☒ For each value of x, the inner loop iterates y from 1 to 9, inserting (y, 'in y loop') into tempp.
☒ After each inner loop completes, the outer loop inserts (x, 'in x loop') into tempp.
☒ The variable y is reset to 1 after each outer loop iteration to ensure the inner loop runs again for the new value of x.
☒ When x reaches 10, the procedure terminates.

Program 3

(Interview question)

```
delete from tempp;    drop procedure abc2;
```

```
delimiter //
create procedure abc2()
begin
    declare x int default 1;
    declare y int default 1;
    while x < 10 do
        while y < x do
            insert into tempp values(y, 'in y loop');
            set y = y + 1;
        end while;
        insert into tempp values(x, 'in x loop');
        set x = x + 1;
    end while;
end; //
delimiter ;
```

```
call abc2;
```

```
select * from tempp;
```

```
+------+-----------+
| fir  | sec       |
+------+-----------+
|    1 | in x loop |
|    1 | in y loop |
|    2 | in x loop |
|    2 | in y loop |
|    3 | in x loop |
|    3 | in y loop |
|    4 | in x loop |
|    4 | in y loop |
|    5 | in x loop |
|    5 | in y loop |
|    6 | in x loop |
|    6 | in y loop |
|    7 | in x loop |
|    7 | in y loop |
|    8 | in x loop |
|    8 | in y loop |
|    9 | in x loop |
+------+-----------+
```

Program Explanation

```
delimiter //
create procedure abc2()
begin
      declare x int default 1;
      declare y int default 1;
      while x < 10 do
            while y < x do
                  insert into tempp values(y, 'in y loop');
                  set y = y + 1;
            end while;
      insert into tempp values(x, 'in x loop');
      set x = x + 1;
      end while;
end; //
delimiter ;
```

```
+------+-----------+
| fir  | sec       |
+------+-----------+
|    1 | in x loop |
|    1 | in y loop |
|    2 | in x loop |
|    2 | in y loop |
|    3 | in x loop |
|    3 | in y loop |
|    4 | in x loop |
|    4 | in y loop |
|    5 | in x loop |
|    5 | in y loop |
|    6 | in x loop |
|    6 | in y loop |
|    7 | in x loop |
|    7 | in y loop |
|    8 | in x loop |
|    8 | in y loop |
|    9 | in x loop |
+------+-----------+
```

☒ **DECLARE x INT DEFAULT 1**: Declares a local variable x and initializes it to 1.
☒ **DECLARE y INT DEFAULT 1**: Declares a local variable y and initializes it to 1.

Outer WHILE Loop

```
WHILE x < 10 DO
    ...
    SET x = x + 1;
END WHILE;
```

- **Condition**: The loop runs as long as `x` is less than 10.
- **SET x = x + 1**: After executing the inner loop and the outer loop's insert statement, increments `x` by 1.

Inner WHILE Loop

```
WHILE y < x DO
    INSERT INTO tempp VALUES (y, 'in y loop');
    SET y = y + 1;
END WHILE;
```

- **Condition**: The loop runs as long as `y` is less than `x`.
- **INSERT INTO tempp VALUES (y, 'in y loop')**: Inserts a row into the `tempp` table with the current value of `y` and the string 'in y loop'.
- **SET y = y + 1**: Increments `y` by 1 after each iteration of the inner loop.

Outer Loop Insert

```
INSERT INTO tempp VALUES (x, 'in x loop');
```

- **INSERT INTO tempp VALUES (x, 'in x loop')**: After the inner loop finishes, inserts a row into the `tempp` table with the current value of `x` and the string 'in x loop'.

Flow Explanation

☒ **Initialization**:
- x is initialized to 1.
- y is initialized to 1.

☒ **Outer WHILE Loop**:
Runs while x is less than 10.
- **Inner WHILE Loop**: Runs while y is less than x.
  - Inserts (y, 'in y loop') into tempp.
  - Increments y by 1.
- When y is no longer less than x, the inner loop exits.
- Inserts (x, 'in x loop') into tempp.
- Increments x by 1.
- **Note**: y is not reset between iterations of the outer loop. Thus, y continues to increment from its last value.

☒ **Termination**:
The outer loop stops when x reaches 10.

Example Run

Initial state:
x = 1
y = 1
First Outer Loop Iteration (x = 1):
- Inner loop does not execute because y is not less than x.
- Insert (1, 'in x loop').
- x becomes 2.

Second Outer Loop Iteration (x = 2):
- Inner loop (y = 1):
- Insert (1, 'in y loop').
- y becomes 2.
- Inner loop (y = 2): Exits because y is not less than x.
- Insert (2, 'in x loop').
- x becomes 3.

Third Outer Loop Iteration (x = 3):
- Inner loop (y = 2):
- Insert (2, 'in y loop').
- y becomes 3.
- Inner loop (y = 3): Exits because y is not less than x.
- Insert (3, 'in x loop').
- x becomes 4.

This pattern continues until x reaches 10.

The variable `y` is not reset within the outer loop, causing it to continue incrementing across iterations of the outer loop. If you want `y` to be reset at the beginning of each outer loop iteration, you should add `SET y = 1;` at the start of the outer loop:

Repeat loop

- Similar to Do…. while loop
- There is no condition to enter the loop, but there is a condition to exit the loop
- It will execute at least once
- **Uses: -**
    - Outerjoin (Master-Detail Report) (Parent-Child Report)

Syntax

REPEAT
    ……;
    ……;
UNTIL expression
END REPEAT;

Program

```
delete from tempp;    drop procedure abc;
```

```
delimiter //
create procedure abc()
begin
    declare x int default 1;
    repeat
        insert into tempp values(x, 'in x loop');
        set x = x + 1;
    until x > 5
    end repeat;
end; //
delimiter ;

call abc;

select * from tempp;
```

```
+------+-----------+
| fir  | sec       |
+------+-----------+
|    1 | in x loop |
|    2 | in x loop |
|    3 | in x loop |
|    4 | in x loop |
|    5 | in x loop |
+------+-----------+
```

Loop, Leave and Iterate statements

- Leave statement allows you to exit the loop (similar to the 'break' command of 'C' programming)
- Iterate statement allows you to skip the entire code under it and start a new iteration (similar to 'continue' statement of 'C' programming)
- Loop statement executes a block of code repeatedly with an additional flexibility of using a loop label

Program

```
delete from tempp;    drop procedure abc;
```

```
delimiter //
create procedure abc()
begin
    declare x int default 1;
    pqr_loop:loop
        if x > 10 then
            leave pqr_loop;
        end if;
        set x = x + 1;
        if mod(x,2) != 0 then
            iterate pqr_loop;
        else
            insert into tempp values(x, 'in x loop');
        end if;
    end loop;
end; //
delimiter ;

call abc;

select * from tempp;
```

```
+------+-----------+
| fir  | sec       |
+------+-----------+
|    2 | in x loop |
|    4 | in x loop |
|    6 | in x loop |
|    8 | in x loop |
|   10 | in x loop |
+------+-----------+
```

Global Variables: -

```
mysql> set @x = 10;

mysql> set @x from dual;          --→10
```

- To create a global variable, no need to declare
- You can name it and initialize it simultaneously
- Global variables can be used in SQL commands, MySQL-PL programs, and can be used in front-end software also
- Global variables remain in the Server RAM till you exit (end of session)

MySQL – PL – Sub-blocks(block within block)

Program

```
delete from tempp;    drop procedure abc;
```

```
delimiter //
create procedure abc()
begin
    declare x int default 1;                 /* GLOBAL VARIABLE */
    insert into tempp values(x, 'before sub');
    begin
        declare y int default 2;            /* LOCAL VARIABLE */
        insert into tempp values(y, 'in sub');
    end;
    insert into tempp values(x, 'after sub');
end; //
delimiter ;

call abc;

select * from tempp;
```

```
+------+------------+
| fir  | sec        |
+------+------------+
|    1 | before sub |
|    2 | in sub     |
|    1 | after sub  |
+------+------------+
```

- main block cannot access variable of sub-block (form of Encapsulation) (Data hiding)
- sub-block can access variable of main block

Program

```
delete from tempp;    drop procedure abc;
```

```
delimiter //
create procedure abc()
begin
    declare x int default 1;                 /* GLOBAL VARIABLE */
    insert into tempp values(x, 'before sub');
    begin
        declare y int default 2;            /* LOCAL VARIABLE */
        insert into tempp values(x, 'in sub');
    end;
    insert into tempp values(x, 'after sub');
end; //
delimiter ;

call abc;

select * from tempp;
```

```
+------+------------+
| fir  | sec        |
+------+------------+
|    1 | before sub |
|    1 | in sub     |
|    1 | after sub  |
+------+------------+
```

Program

```
delete from tempp;    drop procedure abc;
```

```
delimiter //
create procedure abc()
begin
    declare x int default 1;                 /* GLOBAL VARIABLE */
    insert into tempp values(x, 'before sub');
    begin
        declare y int default 2;            /* LOCAL VARIABLE */
        set x = x + y;
        insert into tempp values(x, 'in sub');
    end;
    insert into tempp values(x, 'after sub');
end; //
delimiter ;

call abc;

select * from tempp;
```

```
+------+------------+
| fir  | sec        |
+------+------------+
|    1 | before sub |
|    3 | in sub     |
|    3 | after sub  |
+------+------------+
```

- Sub-block can modify the value of main block's variable, and it will affect the remainder of program

Program

```
delete from tempp;    drop procedure abc;
```

```
delimiter //
create procedure abc()
begin
    declare x int default 1;                 /* GLOBAL VARIABLE */
    insert into tempp values(x, 'before sub');
    begin
        declare x int default 3;            /* LOCAL VARIABLE */
        insert into tempp values(x, 'in sub');
    end;
    insert into tempp values(x, 'after sub');
end; //
delimiter ;

call abc;

select * from tempp;
```

```
+------+------------+
| fir  | sec        |
+------+------------+
|    1 | before sub |
|    3 | in sub     |
|    1 | after sub  |
+------+------------+
```

- If you declare a variable with the same name in both the blocks, then higher priority is given to LOCAL variable

Feature of OOPS: -
Benefit of having variable with the same name in both the blocks: -

1. Code Reusability
2. Reduces coding
3. Standardization of code

Program

```
delete from tempp;    drop procedure abc;
```

```
set @x = 1;
delimiter //
create procedure abc()
begin
    insert into tempp values(@x, 'before sub');
    begin
        declare x int default 3;            /* LOCAL VARIABLE */
        insert into tempp values(x, 'in sub');
        insert into tempp values(@x, 'in sub');
    end;
    insert into tempp values(@x, 'after sub');
end; //
delimiter ;

call abc;

select * from tempp;
```

```
+------+------------+
| fir  | sec        |
+------+------------+
|    1 | before sub |
|    3 | in sub     |
|    1 | in sub     |
|    1 | after sub  |
+------+------------+
```

DAY 12

MySQL - PL – Cursors (V. imp)(Most Imp)

```
CREATE TABLE EMP (
    EMPNO INT PRIMARY KEY,
    ENAME VARCHAR(15),
    SAL INT,
    DEPTNO INT
);

-- Insert data into the EMP table
INSERT INTO EMP (EMPNO, ENAME, SAL, DEPTNO)
VALUES
(1, 'A', 5000, 1),
(2, 'B', 6000, 1),
(3, 'C', 7000, 1),
(4, 'D', 9000, 2),
(5, 'E', 8000, 2);
```

```
drop table emp;
```

EMP			
EMPNO	ENAME	SAL	DEPTNO
1	A	5000	1
2	B	6000	1
3	C	7000	1
4	D	9000	2
5	E	8000	2

```
create table tempp
(
fir int,
sec char(15)
);
```

```
drop table tempp;
```

TEMPP	
FIR	SEC

- Cursors are present in all RDBMS, some of the DBMS, and some of the front-end software also
- Cursor is a type of a variable

```
declare x int;
deeclare hra float;
select sal into x from emp where empno = 1;
set hra = x*0.4;
insert into tempp……;
select sal into x from emp where empno = 2;
set hra = x*0.4;
insert into tempp……;
select sal into x from emp where empno = 3;
set hra = x*0.4;
insert into tempp……;
select sal into x from emp where empno = 4;
set hra = x*0.4;
insert into tempp……;
select sal into x from emp where empno = 5;
set hra = x*0.4;
insert into tempp……;
```

```
declare x int;
declare hra float;
declare z int default 1;
while z < 6 do
         select sal into x from emp where empno = z;
         set hra = x*0.4;
         insert into tempp……;
         set z = z+1;
end while;
```

- Cursor can store multiple rows
- Cursor is similar to 2D array
- Cursor is used for processing multiple rows
- used for handlng multiple rows
- used for storing the data temporarily
- Cursor is similar to a 2D array
- Cursor is based on SELECT statement
- Cursor is READ_ONLY variable
- The data that is present in the cursor, it cannot be manipulated
- You will have to fetch 1 row at a time into some intermediate variables, and do your processing with those variables
- You can only fetch sequntiall (top to bottom)
- YOU CANNOT FETCH BAACKWARDS IN A MySQL CURSOR

A cursor in MySQL is an iterator used to store the variables in a stored procedure. It helps us to iterate through a result given by a query. Results are in the form of a set of rows, and MySQL cursor help to process each row one at a time.Typically, you use cursors within stored procedures, triggers, and functions where you need **to process individual rows returned by a query one at a time**.

```
-- declare a cursor
DECLARE cursor_name CURSOR FOR
SELECT column1, column2
FROM your_table
WHERE your_condition;
-- open the cursor
OPEN cursor_name;
-- fetch the data
FETCH cursor_name INTO variable1, variable2;
-- process the data
-- close the cursor
CLOSE cursor_name;
```

Let us look at all the steps below to use the cursor in MySQL.

Step 1: Firstly, we declare a cursor by using the DECLARE statement.

```
DECLARE cursor_name CURSOR FOR SELECT statement;
or
DECLARE cursor_name CURSOR FOR
SELECT column1, column2
FROM your_table
WHERE your_condition;
```

- The cursor declaration must come after any variable declaration. If you declare a cursor before the variable declarations, MySQL will issue an error.
- Additionally, a cursor must always associate with a SELECT statement.

Step 2: Now, we will open the cursor using the OPEN statement. The OPEN statement is used to open a cursor in MySQL. This will run the SELECT statement that was specified in the previous step and will prepare the cursor to fetch the first row of data.

```
OPEN cursor_name;
```

OPEN statement initialises the cursor, which is why it is necessary to OPEN the cursor before fetching the data from the result set.

Step 3: After opening the cursor, we will fetch the row or column into a variable list.

```
FETCH cursor_name INTO variables list;
or
FETCH cursor_name INTO variable1, variable2;
```

The FETCH statement is used to get the next row to which the cursor will point.

Step 4: As we have stored the data in a variable list. Now, we will close the cursor by using the CLOSE statement. It helps us to release the memory associated with the cursor.

```
CLOSE cursor_name;
```

Note: While working with MySQL cursor, we should also use the NOT FOUND condition to handle the condition when the next row is not present.

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;
```

In the above statement, the finished keyword is used to define the condition that we have reached the end of the result set, and we should not continue after this.

Program

```
delete from tempp;    drop procedure abc;
```

```
delimiter //
create procedure abc()
begin
    declare a int;
    declare b varchar(15);
    declare c int;
    declare d int;
    declare x int default 0;
    declare c1 cursor for select * from emp;
    open c1;
    while x < 5 do
        fetch c1 into a, b, c, d;
        /* processing e.g., set hra = c*0.4, etc. */
        insert into tempp values(a, b);
        set x = x+1;
    end while;
    close c1;
end; //
delimiter ;

call abc;

select * from emp;

select * from tempp;
```

/* CURSOR DECLARATION / DEFINITION
(AT THIS POINT, THE CURSOR DOES NOT CONTAIN DATA) */

/*THIS WILL OPEN THE CURSOR C1, IT WILL EXECUTE
THE SELECT STATEMENT, AND IT WILL POPULATE THE CURSOR C1*/

Try for x < 3, and x < 10

/* WILL FETCH THE NEXT ROW */

/*THIS WILL CLOSE THE CURSOR C1 AND IT WILL FREE THE SERVER RAM*/

```
mysql> select * from emp;
+-------+-------+------+--------+
| EMPNO | ENAME | SAL  | DEPTNO |
+-------+-------+------+--------+
|     1 | A     | 5000 |      1 |
|     2 | B     | 6000 |      1 |
|     3 | C     | 7000 |      1 |
|     4 | D     | 9000 |      2 |
|     5 | E     | 8000 |      2 |
+-------+-------+------+--------+
```

```
mysql> select * from tempp;
+------+------+
| fir  | sec  |
+------+------+
|    1 | A    |
|    2 | B    |
|    3 | C    |
|    4 | D    |
|    5 | E    |
+------+------+
```

Program

```
delete from tempp;    drop procedure abc;
```

```
ALTER TABLE emp ADD COLUMN hra_calc FLOAT;
```

```
delimiter //

create procedure abc()
begin
    declare a int;
    declare b varchar(15);
    declare c int;
    declare d int;
    declare hra_calc float;
    declare x int default 0;
    declare c1 cursor for select empno, ename, sal, deptno from emp;

    open c1;

    while x < 5 do
        fetch c1 into a, b, c, d;
        set hra_calc = c * 0.4;
        update emp set hra_calc = hra_calc where empno = a;
        insert into tempp values (a, b);
        set x = x + 1;
    end while;

    close c1;
end; //
delimiter ;

call abc;
select * from emp;
select * from tempp;
```

The FETCH statement to fetch empno into a, ename into b, sal into c, and deptno into d.

```
mysql> select * from emp;
+-------+-------+------+--------+-----------+
| EMPNO | ENAME | SAL  | DEPTNO | hra_calc  |
+-------+-------+------+--------+-----------+
|     1 | A     | 5000 |      1 |      2000 |
|     2 | B     | 6000 |      1 |      2400 |
|     3 | C     | 7000 |      1 |      2800 |
|     4 | D     | 9000 |      2 |      3600 |
|     5 | E     | 8000 |      2 |      3200 |
+-------+-------+------+--------+-----------+
```

```
mysql> select * from tempp;
+------+------+
| fir  | sec  |
+------+------+
|    1 | A    |
|    2 | B    |
|    3 | C    |
|    4 | D    |
|    5 | E    |
+------+------+
```

Program

```
delete from tempp;    drop procedure abc;
```

```
delimiter //
create procedure abc()
begin
    declare a int;
    declare b varchar(15);
    declare c int;
    declare d int;
    declare x int default 0;
    declare y int;
    declare c1 cursor for select * from emp;
    select count(*) into y from emp;
    open c1;
    while x < y do
            fetch c1 into a, b, c, d;
            insert into tempp values (a, b);
            set x = x + 1;
    end while;
    close c1;
end; //
delimiter ;

call abc;

select * from emp;

select * from tempp;
```

```
mysql> select * from tempp;
+------+------+
| fir  | sec  |
+------+------+
|    1 | A    |
|    2 | B    |
|    3 | C    |
|    4 | D    |
|    5 | E    |
+------+------+
```

```
mysql> select * from emp;
+-------+-------+------+--------+
| EMPNO | ENAME | SAL  | DEPTNO |
+-------+-------+------+--------+
|     1 | A     | 5000 |      1 |
|     2 | B     | 6000 |      1 |
|     3 | C     | 7000 |      1 |
|     4 | D     | 9000 |      2 |
|     5 | E     | 8000 |      2 |
+-------+-------+------+--------+
```

- **Declare a CONTINUE handler for NOT FOUND event**
- NOT FOUND is a cursor attribute, it returns a boolean TRUE value if the last fetch was unsuccessful, and a boolean FALSE value if the last fetch was successful
- continue handler is pre-defined exception
- declare continue handler for not found event → this exception is raised automatically for not found event

Program

```
delete from tempp;    drop procedure abc;
```

```
delimiter //
create procedure abc()
begin
    declare a int;
    declare b varchar(15);
    declare c int;
    declare d int;
    declare y int default 0;
    declare c1 cursor for select * from emp;
    declare continue handler for not found set y = 1;
    open c1;
    cursor_c1_loop:loop
            fetch c1 into a, b, c, d;
            if y = 1 then
                    leave cursor_c1_loop;
            end if;
            insert into tempp values (a, b);
    end loop cursor_c1_loop;
    close c1;
end; //
delimiter ;

call abc;

select * from emp;

select * from tempp;
```

```
mysql> select * from tempp;
+------+------+
| fir  | sec  |
+------+------+
|    1 | A    |
|    2 | B    |
|    3 | C    |
|    4 | D    |
|    5 | E    |
+------+------+
```

```
mysql> select * from emp;
+-------+-------+------+--------+
| EMPNO | ENAME | SAL  | DEPTNO |
+-------+-------+------+--------+
|     1 | A     | 5000 |      1 |
|     2 | B     | 6000 |      1 |
|     3 | C     | 7000 |      1 |
|     4 | D     | 9000 |      2 |
|     5 | E     | 8000 |      2 |
+-------+-------+------+--------+
```

Program

```
delete from tempp;    drop procedure abc;
```

```
delimiter //
create procedure abc()
begin
    declare a int;
    declare b varchar(15);
    declare c int;
    declare d int;
    declare y int default 0;
    declare c1 cursor for select * from emp where deptno = 1;
    declare continue handler for not found set y = 1;
    open c1;
    cursor_c1_loop:loop
            fetch c1 into a, b, c, d;
            if y = 1 then
                    leave cursor_c1_loop;
            end if;
            insert into tempp values (a, b);
    end loop cursor_c1_loop;
    close c1;
end; //
delimiter ;

call abc;

select * from emp;

select * from tempp;
```

```
mysql> select * from tempp;
+------+------+
| fir  | sec  |
+------+------+
|    1 | A    |
|    2 | B    |
|    3 | C    |
+------+------+
```

```
mysql> select * from emp;
+-------+-------+------+--------+
| EMPNO | ENAME | SAL  | DEPTNO |
+-------+-------+------+--------+
|     1 | A     | 5000 |      1 |
|     2 | B     | 6000 |      1 |
|     3 | C     | 7000 |      1 |
|     4 | D     | 9000 |      2 |
|     5 | E     | 8000 |      2 |
+-------+-------+------+--------+
```

Program: Using parameter

```
delimiter //
create procedure abc(dd int)
begin
    ……;
    declare y int default 0;
    declare c1 cursor for select * from emp where deptno = dd;
    ……;
    open c1;
    cursor_c1_loop:loop
            ……;
    end loop cursor_c1_loop;
    close c1;
end; //
delimiter ;

mysql> call abc(1);
mysql> call abc(2);
```

Uses of Cursors: -

a) used for storing / processing multiple rows
b) used for locking the rows manually

To lock the rows manually: -

- You require SELECT statement with a FOR UPDATE clause
- You declare a cursor whose SELECT statement has a FOR UPDATE clause
- Then you open and close the cursor

```
delimiter //
create procedure abc()
begin
    declare c1 cursor for select * from emp for update;
    open c1;
    close c1;
end; //
delimiter ;

mysql> call abc();
```

- Table will remain locked till you Rollback or Commit
- LOCKS ARE AUTOMATICALLY RELEASED WHEN YOU ROLLBACK OR COMMIT

Parameters

- You can pass parameter to a procedure

Parameters are of 3 types: -

1. **IN** (by default)
   - Read only
   - Value of parameter cannot be changed inside the procedure
   - You can pass a constant, variable, and an expression
   - Call by value
   - FASTEST in terms of processing speed
   - If you don't want to return a value, then choose IN parameter

```
delete from tempp;    drop procedure abc;
```

```
delimiter //
create procedure abc(in y int)
begin
    insert into tempp values(y, 'inside abc');
end; //
delimiter ;
```

```
delimiter //
create procedure pqr()
begin
    declare x int default 10;
    call abc(5);
    call abc(x);
    call abc(2*x + 5);
end; //
delimiter ;

call pqr();
select * from tempp;
```

```
mysql> select * from tempp;
+------+------------+
| fir  | sec        |
+------+------------+
|    5 | inside abc |
|   10 | inside abc |
|   25 | inside abc |
+------+------------+
```

```
call abc(5); /* Passing a constant */
select * from tempp;
```

```
mysql> select * from tempp;
+------+------------+
| fir  | sec        |
+------+------------+
|    5 | inside abc |
+------+------------+
```

```
set @x = 10;
call abc(@x); /* Passing a variable */
select * from tempp;
```

```
mysql> select * from tempp;
+------+------------+
| fir  | sec        |
+------+------------+
|    5 | inside abc |
|   10 | inside abc |
+------+------------+
```

```
set @x = 10;
call abc(2*@x+5); /* Passing an expression */
select * from tempp;
```

```
mysql> select * from tempp;
+------+------------+
| fir  | sec        |
+------+------------+
|    5 | inside abc |
|   10 | inside abc |
|   25 | inside abc |
+------+------------+
```

2. **OUT**
   - Write only
   - Can pass variable only (constant and expressions are NOT ALLOWED)
   - Call by reference
   - You can return a value indirectly
   - If you want to return a value indirectly and Security is important e.g., username, password, OTP, etc.
   - Used on public network e.g., Internet

```
delete from tempp;    drop procedure abc;   drop procedure pqr;
```

```
delimiter //
create procedure abc(out y int)
begin
        set y = 100;
end; //
delimiter ;
```

```
delimiter //
create procedure pqr()
begin
    declare x int default 10;
    insert into tempp values(x, 'before abc');
    call abc(x);
    insert into tempp values(x, 'after abc');
end; //
delimiter ;

call pqr();
select * from tempp;
```

```
mysql> select * from tempp;
+------+------------+
| fir  | sec        |
+------+------------+
|   10 | before abc |
|  100 | after abc  |
+------+------------+
```

```
set @x = 10;
select @x from dual;
```

```
+------+
| @x   |
+------+
|   10 |
+------+
```

```
call abc(@x); /* Address is passed not value */
select @x from dual;
```

```
+------+
| @x   |
+------+
|  100 |
+------+
```

3. INOUT

- Read and write
- Can pass variables only
- Call by reference
- You can return a value indirectly
- BEST FUNCTIONALITY
- If you want to return a value indirectly and Security is not important
- Used on local network, e.g., home network, CDAC

```
delete from tempp;    drop procedure abc;   drop procedure pqr;
```

```
delimiter //
create procedure abc(inout y int)
begin
        set y = y*y*y;
end; //
delimiter ;
```

```
delimiter //
create procedure pqr()
begin
    declare x int default 10;
    insert into tempp values(x, 'before abc');
    call abc(x);
    insert into tempp values(x, 'after abc');
end; //
delimiter ;

call pqr();
select * from tempp;
```

```
mysql> select * from tempp;
+------+------------+
| fir  | sec        |
+------+------------+
|   10 | before abc |
| 1000 | after abc  |
+------+------------+
```

```
set @x = 10;
select @x from dual;
```

```
+------+
| @x   |
+------+
|   10 |
+------+
```

```
call abc(@x); /* Address is passed not value */
select @x from dual;
```

```
+------+
| @x   |
+------+
| 1000 |
+------+
```

MySQL – STORED OBJECTS

- Objects that are stored in the database
- E.g., CREATE……. table, indexes, views, stored procedures
- Anything that you do it with CREATE command is a stored object

STORED FUNCTIONS

- Routine (set of commands) that returns a value directly and compulsorily
- Unlike a procedure a function cannot be called itself, because a function returns a value, and that value has to be stored somewhere; and therefore, it has to equated with a variable, or it has to be a part of some expression
- Can be called through MySQL-PL program (stored procedure), Java, MS .Net, etc.
- Global functions
- Can be called from any front-end software
- Stored in the database in the COMPILE FORMAT
- Hence the execution will be very fast
- Hiding source code from end user
- Within the function all MySQL-PL commands are allowed
- Functions can contain local variables, cursors, IF statement, loops, sub-blocks etc.
- Stored procedure can call stored function
- Stored function can call stored procedure
- One function can call another function
- Function can call itself (known as Recursion)
- You can pass parameters to a function
- IN parameters only
- OVERLOADING OF STORED FUNCTIONS IS NOT ALLOWED because it is a stored object
- You cannot create 2 or more functions with the same name even if the NUMBER of parameters passed is different or the DATATYPE of parameters passed is different.

Functions are of 2 types: -

1. **Deterministic**
2. **Not-Deterministic**

- For the same input parameters, if the stored functions return the same result, then it is considered *deterministic*, and otherwise the stored function is *not deterministic*
- You have to decide whether a stored function is deterministic or not
- If you declare it incorrectly, the stored function may produce an unexpected, or the available optimization is not used which degrades the performance
- Mostly functions are deterministic; if the function contains sysdate() or now() and they are used for processing, then the function is not deterministic

Program `delete from tempp;  drop procedure abc;  drop procedure pqr;`

```
delimiter //
create function abc()
returns int
deterministic
begin
        return 10;
end; //
delimiter ;
```

/*→Read, Compile, Plan, and Store it in the DB in the COMPILED FORMAT*/

/*Function created*/

/* Difference between procedure and function*/
/*Unlike a procedure, a function cannot be called by itself i.e., `call abc();` , because a function returns a value, and that value has to be stored somewhere and, therefore, it has to be equated with a variable, or it has to be a part of an expression*/

```
delimiter //
create procedure pqr()
begin
declare x int;
set x = abc();
insert into tempp values(x, 'after abc');
end; //
delimiter ;

call pqr();

select * from tempp;
```

```
mysql> select * from tempp;
+------+-----------+
| fir  | sec       |
+------+-----------+
|   10 | after abc |
+------+-----------+
```

Program `delete from tempp;  drop function abc;  drop procedure pqr;`

```
delimiter //
create function abc(y int)
returns int
deterministic
begin
        return y*y;
end; //
delimiter ;

delimiter //
create procedure pqr()
begin
declare x int;
set x = abc(10);
insert into tempp values(x, 'after abc');
end; //
delimiter ;

call pqr();

select * from tempp;
```

```
mysql> select * from tempp;
+------+-----------+
| fir  | sec       |
+------+-----------+
|  100 | after abc |
+------+-----------+
```

Interview Question

Q1. What is similarity between procedure and function

- Stored procedure is a stored object, Stored function is also a stored object i.e., both are stored object
- Both of them are in database
- Both of them are in compile format
- Hence the execution is very fast, execution takes place in Server RAM
- Because both of them are in database, both of them are GLOBAL you can call them from anywhere (front-end software)
- You can pass parameters

Q2. What is difference between procedure and function

- Procedure does not return a value unless you call by reference, Function returns a value
- Procedure can have IN, OUT, INOUT parameters, Function can have only IN parameter
- A procedure can be called by itself, a Function cannot be called by itself (because a function returns a value, and that value has to be stored somewhere and, therefore, it has to be equated with a variable, or it has to be a part of some expression)
- **STORED FUNCTION CAN BE CALLED IN SELECT STATEMENT**
- **STORED FUNCTION CAN BE CALLED IN SQL COMMANDS**

```
delimiter //
create function abc(y int)
returns int
deterministic
begin
          return y*y;
end; //
delimiter ;
```

```
select abc(sal) from emp;
select abc(10) from dual;
delete from emp where abc(sal) = 1000000;
```

	STORED PROCEDURE	FUNCTION
1	Supports in, out and in-out parameters, i.e., input and output parameters	Supports only input parameters, no output parameters
2	Stored procedures can call functions as needed	The functions cannot call a stored procedure
3	There is no provision to call procedures from SELECT/ HAVING and WHERE statements	You can call functions from a SELECT statement
4	Transactions can be used in stored procedures	No transactions are allowed
5	Can do exception handling by inserting try/ catch blocks	No provision for explicit exception handling
6	Need not return any value	Must return a result or value to the caller
7	All the database operations like insert, update, delete can be performed	Only SELECT is allowed
8	Can return multiple values	Typically returns only one value
9	Example: - CREATE PROCEDURE proename ….define parameters AS ….query GO	Example: - Pre-defined functions- AVG, COUNT, TRIM, IF, CURDATE, ISNULL, etc. User-defined(custom function)- CREATE FUNCTION funcname(func param1, func param2) RETURN value ….body of the function

```
drop table emp;
```

EMP	
ENAME	SAL
KING	9000

```
CREATE TABLE EMP (
    ENAME VARCHAR(15),
    SAL INT
);

-- Insert data into the EMP table
INSERT INTO EMP (ENAME, SAL)
VALUES
('KING', 9000);
```

```
drop table tempp;
```

TEMPP	
FIR	SEC

```
create table tempp
(
fir int,
sec char(15)
);
```

Program

```
delete from tempp;    drop function abc;    drop procedure pqr;
```

```
delimiter //
create function abc(y int)
returns boolean
deterministic
begin
        if y > 5000 then
            return TRUE;
        else
            return FALSE;
        end if;
end; //
delimiter ;

delimiter //
create procedure pqr()
begin
    declare x int;
    select sal into x from emp where ename = 'KING';
    if abc(x) then
        insert into tempp values(x, '> 5000');
    else
        insert into tempp values(x, '<= 5000');
    end if;
end; //
delimiter ;

call pqr();

select * from tempp;
```

```
mysql> select * from tempp;
+------+--------+
| fir  | sec    |
+------+--------+
| 9000 | > 5000 |
+------+--------+
```

To drop the function: -

```
drop function abc;
```

To see which all functions are created: -

```
show function status;        /*←Shows all functions in all schemes*/
```

To view functions created in database juhu: -

```
show function status where db = 'juhu';
```

To see functions which are starting with letter a: -

```
show function status where name like 'a%';
```

To view the source code of stored function: -

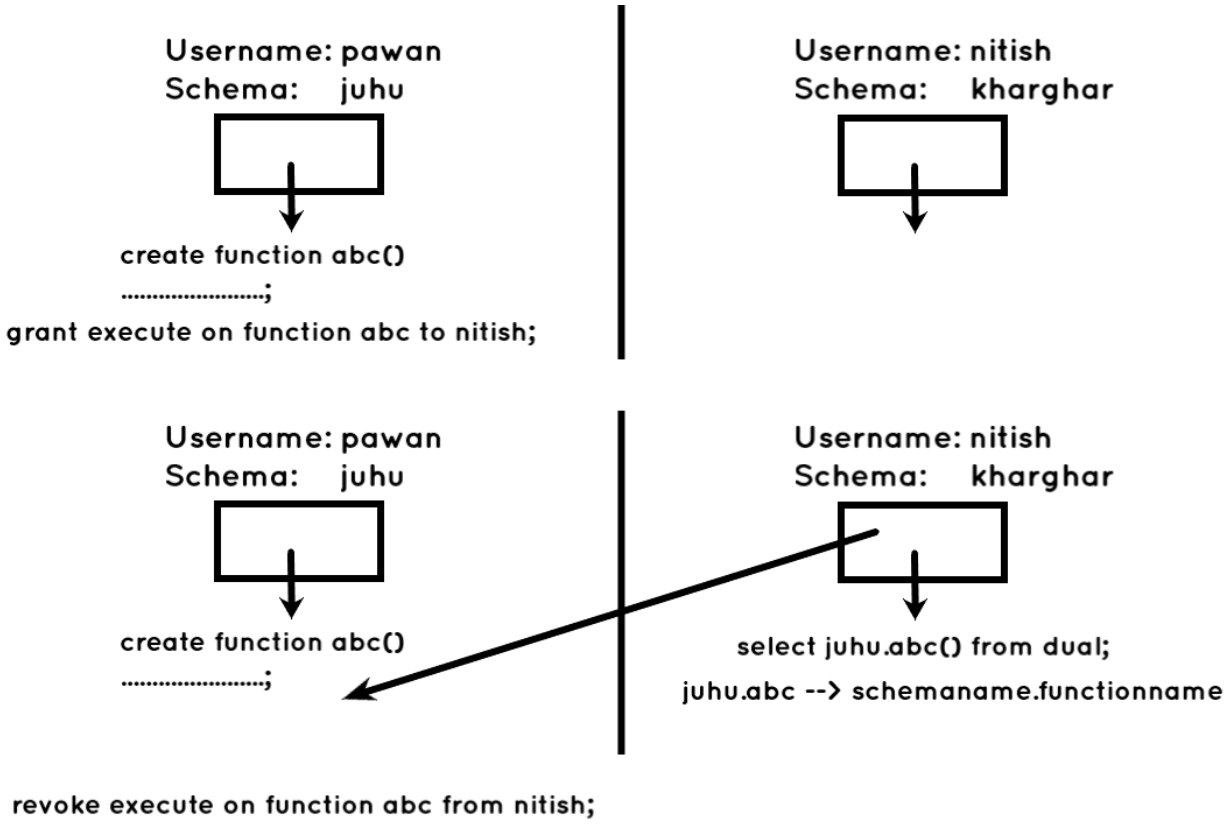
```
show create function abc;
```

To share function with other users: -

```
pawan_mysql>  grant execute on function abc to nitish;
nitish_mysql> select juhu.abc() from dual;
```

To stop sharing of function with other users: -

```
pawan_mysql>  revoke execute on function abc from nitish;
```



DAY 13

MySQL – STORED OBJECTS

- Objects that are stored in the database
- E.g., CREATE ……. tables, indexes, views, stored procedures, stored functions
- Anything that you do with CREATE command

DATABASE TRIGGERS (V. imp)

(A MySQL trigger is a stored program (with queries) which is executed automatically to respond to a specific event such as insertion, updation or deletion occurring in a table.)

- Present in some of the RDBMS
- MS Access, Paradox, Vatcom SQL
- Routine (set of commands) that gets executed AUTOMATICALLY whenever some EVENT takes place
- Triggers are written on tables
- Events are: -
  - Before Insert, After Insert
  - Before Delete, After Delete
  - Before Update, After Update

```
drop table emp;
```

EMP

ENAME	SAL	DEPTNO
A	5000	1
B	5000	1
C	5000	1
D	3000	2
E	3000	2

```
create table emp(
ENAME varchar(20),
SAL int,
DEPTNO int);

insert into emp(ENAME, SAL, DEPTNO)
values
('A',5000,1),
('B',5000,1),
('C',5000,1),
('D',3000,2),
('E',3000,2);
```

DEPTOT

DEPTNO	SALTOT
1	15000
2	6000

here,
DEPTOT = department total
SALTOT = salary total

```
create table deptot(
DEPTNO int,
SALTOT int
);

insert into
deptot(DEPTNO,SALTOT)
values
(1,15000),
(2,6000);
```

```
drop table tempp;
```

TEMPP

FIR	SEC

```
create table tempp
(
fir int,
sec char(15)
);
```

```
select deptno, sum(sal) from emp
group by deptno;
```

```
+--------+----------+
| deptno | sum(sal) |
+--------+----------+
|      1 |    15000 |
|      2 |     6000 |
+--------+----------+
```

```
select * from deptot;
```

```
+--------+--------+
| DEPTNO | SALTOT |
+--------+--------+
|      1 |  15000 |
|      2 |   6000 |
+--------+--------+
```

INSERT event

```
delimiter //
create trigger abc
before insert
on emp for each row
begin
    insert into tempp values(1, 'inserted');
    /*commit*/
end; //
delimiter ;

-- Trigger created

/*Read, Compile, Plan, Store it in the database in the COMPILED FORMAT*/

/*BEFORE INSERT ON emp: Specifies that the trigger should fire before an insert operation on the emp table.*/

insert into emp values('F',3000,2);
```

```
select * from emp;
```

```
mysql> select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 5000 |      1 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
| D     | 3000 |      2 |
| E     | 3000 |      2 |
| F     | 3000 |      2 |
+-------+------+--------+
```

```
select * from tempp;
```

```
mysql> select * from tempp;
+------+----------+
| fir  | sec      |
+------+----------+
|    1 | inserted |
+------+----------+
```

- All the triggers are at Server level; you can perform the DML operations using MySQL Command Line Client, MySQL Workbench, Java, MS .Net, etc. or any other front-end software, the triggers will always execute
- Within the trigger, all MySQL-PL statements allowed, e.g., declare variables, IF statement, loops, cursors, sub-blocks, etc
- Rollback and Commit not allowed inside the trigger
- Rollback or Commit has to be specified AFTERWARDS, at the end of Transaction
- Whether you Rollback or Commit AFTERWARDS, the data will always be consistent
- If the DML operation on the table fails, then it will cause the event to fail, and then the trigger changes are automatically Rolled back
- If the trigger fails, then it will cause the event to fail, and then the DML operation on the table is automatically Rolled back
- YOUR DATA WILL ALWAYS BE CONSISTENT
- In MySQL, all triggers are at Row level (will execute once for each row inserted)
- In MySQL you can have maximum 6 triggers per table

USES: -

- Maintain logs (audit trails) of insertions
  (AFTER insert trigger is recommended)

In MySQL, all triggers are at Row level (will execute once for each row inserted)

EMP

ENAME	SAL	DEPTNO
A	5000	1
B	5000	1
C	5000	1
D	3000	2
E	3000	2

EMP2

ENAME	SAL	DEPTNO
F	5000	1
G	5000	2
H	5000	1
I	3000	2
J	3000	2

```
delimiter //
create trigger abc
before insert
on emp for each row
begin
     insert into tempp values(1, 'inserted');
end; //
delimiter ;

insert into emp select * from emp2;
```

```
mysql> select * from tempp;
+------+----------+
| fir  | sec      |
+------+----------+
|    1 | inserted |
|    1 | inserted |
|    1 | inserted |
|    1 | inserted |
|    1 | inserted |
+------+----------+
```

EMP

ENAME	SAL	DEPTNO
A	5000	1
B	5000	1
C	5000	1
D	3000	2
E	3000	2
F	5000	1
G	5000	2
H	5000	1
I	3000	2
J	3000	2

5 rows inserted from emp2 table

Program

```
drop trigger abc;     delete from tempp;
```

```
select * from emp;
```

```
mysql> select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 5000 |      1 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
| D     | 3000 |      2 |
| E     | 3000 |      2 |
| F     | 3000 |      2 |
+-------+------+--------+
```

```
delimiter //
create trigger abc
before insert
on emp for each row
begin
     insert into tempp values(new.sal, new.ename);
end; //
delimiter ;

insert into emp values('G',6000,1);
```

```
select * from tempp;
```

```
mysql> select * from tempp;
+------+------+
| fir  | sec  |
+------+------+
| 6000 | G    |
+------+------+
```

```
select * from emp;
```

```
mysql> select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 5000 |      1 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
| D     | 3000 |      2 |
| E     | 3000 |      2 |
| F     | 3000 |      2 |
| G     | 6000 |      1 |
+-------+------+--------+
```

- new.ename, new.sal, new.deptno are MySQL created variables

USES: -

- maintain multiple copies of the table in the event of INSERT
- automatic data duplication, data mirroring, data replication, concept of standby database, primary and secondary servers, Data Guard
- maintain SHADOW tables in the event of INSERT
- Data cleansing BEFORE insert
- Validations BEFORE insert
- Dynamic Default value BEFORE insert

Program: Dynamic Default value BEFORE insert

```
drop trigger abc;    delete from tempp;
```

```
select * from emp;
```

```
mysql> select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 5000 |      1 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
| D     | 3000 |      2 |
| E     | 3000 |      2 |
| F     | 3000 |      2 |
| G     | 6000 |      1 |
+-------+------+--------+
```

```
delimiter //
create trigger abc
before insert
on emp for each row
begin
     if new.deptno = 1 then
          set new.sal = 5000;
     else
          set new.sal = 3000;
end if;
end; //
delimiter ;

insert into emp(ename,sal,deptno)
values
('H', 0, 1),
('I',NULL, 3);
```

```
select * from emp;
```

- Though `sal` values of new row inserted are 0 and NULL, but while storing them in table, they are stored as per condition of `deptno` given in trigger with default value of either 5000 or 3000

```
if new.deptno = 1 then
          set new.sal = 5000;
     else
          set new.sal = 3000;
```

```
mysql> select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 5000 |      1 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
| D     | 3000 |      2 |
| E     | 3000 |      2 |
| F     | 3000 |      2 |
| G     | 6000 |      1 |
| H     | 5000 |      1 |
| I     | 3000 |      3 |
+-------+------+--------+
```

Program

```
drop trigger abc;     delete from tempp;
```

```
select * from emp;
mysql> select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 5000 |      1 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
| D     | 3000 |      2 |
| E     | 3000 |      2 |
| F     | 3000 |      2 |
| G     | 6000 |      1 |
| H     | 5000 |      1 |
| I     | 3000 |      3 |
+-------+------+--------+
```

```
delimiter //
create trigger abc
before insert
on emp for each row
begin
    update deptot set saltot = saltot + new.sal
    where deptno = new.deptno;
end; //
delimiter ;
```

```
select * from deptot;
+--------+--------+
| DEPTNO | SALTOT |
+--------+--------+
|      1 |  15000 |
|      2 |   6000 |
+--------+--------+
```

```
insert into emp values('J',2000,2);
```

```
                                        (6000)      (2000)
        update deptot set saltot = saltot + new.sal
        where deptno = new.deptno;
                                  (2)
```

```
select * from deptot;
+--------+--------+
| DEPTNO | SALTOT |
+--------+--------+
|      1 |  15000 |
|      2 |   8000 |
+--------+--------+
```

```
select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 5000 |      1 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
| D     | 3000 |      2 |
| E     | 3000 |      2 |
| F     | 3000 |      2 |
| G     | 6000 |      1 |
| H     | 5000 |      1 |
| I     | 3000 |      3 |
| J     | 2000 |      2 |
+-------+------+--------+
```

USES: - Automatic updation of related tables (AFTER insert trigger is recommended)

To drop trigger: -

```
drop trigger abc;
```

- **If you drop the table, then the indexes and triggers are dropped automatically (view, procedure will remain)**

To show all triggers in all schemas: -

```
show triggers;
```

To show all triggers in particular database(juhu): -

```
show triggers from [database_name];
```

```
show triggers juhu;
```

To show detail information of all triggers: -

```
select * from information_schema.triggers;
```

To show detail information of all triggers in 'juhu' schema: -

```
select * from information_schema.triggers;
where trigger_schema = 'juhu';
```

- Within the trigger you can call stored procedure and stored functions

DELETE event

```
drop trigger abc;     delete from tempp;
```

```
select * from emp;
```

```
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 5000 |      1 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
| D     | 3000 |      2 |
| E     | 3000 |      2 |
| F     | 3000 |      2 |
| G     | 6000 |      1 |
| H     | 5000 |      1 |
| I     | 3000 |      3 |
| J     | 2000 |      2 |
+-------+------+--------+
```

```
delimiter //
create trigger pqr
before delete
on emp for each row
begin
    insert into tempp values(1, 'deleted');
end; //
delimiter ;

delete from emp where deptno = 2;
```

```
select * from tempp;
```

```
mysql> select * from tempp;
+------+---------+
| fir  | sec     |
+------+---------+
|    1 | deleted |
|    1 | deleted |
|    1 | deleted |
|    1 | deleted |
+------+---------+
```

```
select * from emp;
```

```
mysql> select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 5000 |      1 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
| G     | 6000 |      1 |
| H     | 5000 |      1 |
| I     | 3000 |      3 |
+-------+------+--------+
```

USES: -

- Maintain logs (audit trails) of deletions AFTER delete trigger is recommended)

```
drop trigger pqr;
drop table tempp;
drop table emp;
drop table deptot;
```

```
create table emp(
ENAME varchar(20),
SAL int,
DEPTNO int);
insert into emp(ENAME,SAL,DEPTNO)
values
('A',5000,1),
('B',5000,1),
('C',5000,1),
('D',3000,2),
('E',3000,2);
```

```
create table deptot(
DEPTNO int,
SALTOT int
);

insert into
deptot(DEPTNO,SALTOT)
values
(1,15000),
(2,6000);
```

```
CREATE  TABLE tempp (
action_id INT,
action_type VARCHAR(50),
executed_by VARCHAR(100),
executed_at DATETIME,
system_time DATETIME
);
```

Program

```
mysql> select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 5000 |      1 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
| D     | 3000 |      2 |
| E     | 3000 |      2 |
+-------+------+--------+
```

```
delimiter //
create trigger pqr
before delete
on emp for each row
begin
    insert into tempp values(1, 'deleted', user(), now(),sysdate());
end; //
delimiter ;
```

```
delete from emp where deptno = 2;
```

```
select * from tempp;
mysql> select * from tempp;
+-----------+-------------+----------------+---------------------+---------------------+
| action_id | action_type | executed_by    | executed_at         | system_time         |
+-----------+-------------+----------------+---------------------+---------------------+
|         1 | deleted     | root@localhost | 2024-08-08 20:01:57 | 2024-08-08 20:01:57 |
|         1 | deleted     | root@localhost | 2024-08-08 20:01:57 | 2024-08-08 20:01:57 |
+-----------+-------------+----------------+---------------------+---------------------+
```

```
select * from emp;
mysql> select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 5000 |      1 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
+-------+------+--------+
```

USES: -

- Maintain logs (audit trails) of deletions (AFTER delete trigger is recommended)

```
drop trigger pqr;    drop trigger abc;
drop table tempp;
drop table emp;
drop table deptot;
```

```
create table emp(
ENAME varchar(20),
SAL int,
DEPTNO int);
insert into emp(ENAME,SAL,DEPTNO)
values
('A',5000,1),
('B',5000,1),
('C',5000,1),
('D',3000,2),
('E',3000,2);
```

```
create table deptot(
DEPTNO int,
SALTOT int
);

insert into
deptot(DEPTNO,SALTOT)
values
(1,15000),
(2,6000);
```

```
create table tempp
(
fir int,
sec char(15)
);
```

Program

```
mysql> select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 5000 |      1 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
| D     | 3000 |      2 |
| E     | 3000 |      2 |
+-------+------+--------+
```

```
delimiter //
create trigger pqr
before delete
on emp for each row
begin
    insert into tempp values(old.sal, old.ename);
end; //
delimiter ;
```

- old.ename, old.sal, old.deptno are MySQL created variables
- if you TRUNCATE the table, then the DELETE triggers on that table will not execute

```
delete from emp where deptno = 2;
```

```
select * from tempp;
+------+------+
| fir  | sec  |
+------+------+
| 3000 | D    |
| 3000 | E    |
+------+------+
```

```
select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 5000 |      1 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
+-------+------+--------+
```

USES:

- Maintain HISTORY tables in the event of delete
  (AFTER delete trigger is recommended)

Program:- Auto Update

```
delimiter //
create trigger abc
before delete
on emp for each row
begin
    update deptot set saltot = saltot - old.sal
    where deptno = old.deptno;
end; //
delimiter ;

delete from emp where ename = 'C';
select * from deptot;
```

```
mysql> select * from deptot;
+--------+--------+
| DEPTNO | SALTOT |
+--------+--------+
|      1 |  10000 |
|      2 |   6000 |
+--------+--------+
```

USES: - Auto-updation of related tables (AFTER delete trigger is recommended)

UPDATE event

```
drop trigger pqr;    drop trigger abc;    drop trigger xyz;
drop table tempp;
drop table emp;
drop table deptot;
```

```
create table emp(
ENAME varchar(20),
SAL int,
DEPTNO int);
insert into emp(ENAME,SAL,DEPTNO)
values
('A',5000,1),
('B',5000,1),
('C',5000,1),
('D',3000,2),
('E',3000,2);
```

```
create table deptot(
DEPTNO int,
SALTOT int
);

insert into
deptot(DEPTNO,SALTOT)
values
(1,15000),
(2,6000);
```

```
create table tempp
(
fir int,
sec char(15)
);
```

Program

EMP		
ENAME	SAL	DEPTNO
A	~~5000~~ 6000	1
B	~~5000~~ 6000	1
C	~~5000~~ 6000	1
D	3000	2
E	3000	2

```
delimiter //
create trigger xyz
before update
on emp for each row
begin
insert into tempp values(1, 'updated');
end; //
delimiter ;
/*Trigger created*/

update emp set sal = 6000
where deptno = 1;
```

DEPTOT	
DEPTNO	SALTOT
1	15000
2	6000

TEMPP	
FIR	SEC
1	updated
1	updated
1	updated

```
delimiter //
create trigger xyz
before update
on emp for each row
begin
    insert into tempp values(1, 'updated');
end; //
delimiter ;
/*Trigger created*/
```

```
update emp set sal = 6000
where deptno = 1;
```

```
select * from tempp;
+------+---------+
| fir  | sec     |
+------+---------+
|    1 | updated |
|    1 | updated |
|    1 | updated |
+------+---------+
```

```
select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 6000 |      1 |
| B     | 6000 |      1 |
| C     | 6000 |      1 |
| D     | 3000 |      2 |
| E     | 3000 |      2 |
+-------+------+--------+
```

- sal column is updated

USES: - Maintain logs (Audit trails) of updations

Cascading triggers

One trigger causes a second trigger to execute, which in turn causes a third trigger to execute, and so on is known as cascading triggers

- In MySQL and Oracle, there is no upper limit on the number of levels for Cascading triggers

Mutating table error

If some of the cascading trigger is going to cause one of the previous triggers to execute, then MySQL will not go into infinite loop; you will get an error Mutating tables, and the entire transaction is automatically Rolled back

```
delimiter //
create trigger xyz
before update
on emp for each row
begin
    insert into tempp values(1, 'updated');
end; //
delimiter ;
```

```
delimiter //
create trigger xyz2
before insert
on tempp for each row
begin
    delete from deptot where ……;
end; //
delimiter ;
```

```
delimiter //
create trigger xyz3
before delete
on deptot for each row
begin
    ……;
end; //
delimiter ;
```

```
drop trigger pqr;    drop trigger abc;    drop trigger xyz;
drop table tempp;
drop table emp;
drop table deptot;
```

```
create table emp(
ENAME varchar(20),
SAL int,
DEPTNO int);
insert into emp(ENAME,SAL,DEPTNO)
values
('A',5000,1),
('B',5000,1),
('C',5000,1),
('D',3000,2),
('E',3000,2);
```

```
create table deptot(
DEPTNO int,
SALTOT int
);

insert into
deptot(DEPTNO,SALTOT)
values
(1,15000),
(2,6000);
```

```
create table tempp
(
fir int,
sec char(15)
);
```

Program

```
delimiter //
create trigger xyz
before update
on emp for each row
begin
    insert into tempp values(old.sal, old.ename);
    insert into tempp values(new.sal, new.ename);
end; //
delimiter ;
```

- old.ename, old.sal, old.deptno, new.ename, new.sal, new.deptno are MySQL created variables

```
update emp set sal = 6000
where deptno = 1;
```

```
select * from tempp;
+------+------+
| fir  | sec  |
+------+------+
| 5000 | A    |
| 6000 | A    |
| 5000 | B    |
| 6000 | B    |
| 5000 | C    |
| 6000 | C    |
+------+------+
```

```
select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 6000 |      1 |
| B     | 6000 |      1 |
| C     | 6000 |      1 |
| D     | 3000 |      2 |
| E     | 3000 |      2 |
+-------+------+--------+
```

USES:

- maintain SHADOW and HISTORY tables in the event of the update (AFTER update trigger is recommended)

```
drop trigger pqr;    drop trigger abc;    drop trigger xyz;
drop table tempp;
drop table emp;
drop table deptot;
```

```
create table emp(
ENAME varchar(20),
SAL int,
DEPTNO int);
insert into emp(ENAME,SAL,DEPTNO)
values
('A',5000,1),
('B',5000,1),
('C',5000,1),
('D',3000,2),
('E',3000,2);
```

```
create table deptot(
DEPTNO int,
SALTOT int
);

insert into
deptot(DEPTNO,SALTOT)
values
(1,15000),
(2,6000);
```

```
create table tempp
(
fir int,
sec char(15)
);
```

Program

```
delimiter //
create trigger xyz
before update
on emp for each row
begin
    update deptot set saltot = saltot - old.sal + new.sal
    where deptno = old.deptno;
end; //
delimiter ;
```

```
                                    (15000)      (5000)       (6000)
update deptot set saltot = saltot - old.sal + new.sal
where deptno = old.deptno;
                    (1)
```

- old.ename, old.sal, old.deptno, new.ename, new.sal, new.deptno are MySQL created variables

```
update emp set sal = 6000
where ename = 'A';
```

```
select * from deptot;
+--------+--------+
| DEPTNO | SALTOT |
+--------+--------+
|      1 |  16000 |
|      2 |   6000 |
+--------+--------+
```

```
select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 6000 |      1 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
| D     | 3000 |      2 |
| E     | 3000 |      2 |
+-------+------+--------+
```

USES: Auto-updation of related tables (AFTER update trigger is recommended)

```
/*To prevent from unnecessary processing(only if sal column is
updated trigger will execute and if any other column is updated
trigger will not execute)*/

delimiter //
create trigger xyz
before update
on emp for each row
begin
    if old.sal <> new.sal then
        update deptot set saltot = saltot - old.sal + new.sal
        where deptno = old.deptno;
    end if;
end; //
delimiter ;
```

Program

```
drop trigger xyz;
```

```
delimiter //
create trigger xyz
before update
on emp for each row
begin
    if old.deptno <> new.deptno or old.sal <> new.sal then
        if old.deptno <> new.deptno then
            update deptot set saltot = saltot - old.sal
            where deptno = old.deptno;
            update deptot set saltot = saltot + new.sal
            where deptno = new.deptno;
        else
        /* IF YOU ARE UPDATING ONLY THE SAL COLUMN */
            update deptot set saltot = saltot - old.sal + new.sal
            where deptno = old.deptno;
        end if;
    end if;
end; //
delimiter ;
```

```
select * from deptot;
+--------+--------+
| DEPTNO | SALTOT |
+--------+--------+
|      1 |  16000 |
|      2 |   6000 |
+--------+--------+
```

```
select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 6000 |      1 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
| D     | 3000 |      2 |
| E     | 3000 |      2 |
+-------+------+--------+
```

```
update emp set deptno = 2
where ename = 'A';
```

```
select * from deptot;
+--------+--------+
| DEPTNO | SALTOT |
+--------+--------+
|      1 |  10000 |
|      2 |  12000 |
+--------+--------+
```

```
select * from emp;
+-------+------+--------+
| ENAME | SAL  | DEPTNO |
+-------+------+--------+
| A     | 6000 |      2 |
| B     | 5000 |      1 |
| C     | 5000 |      1 |
| D     | 3000 |      2 |
| E     | 3000 |      2 |
+-------+------+--------+
```

DAY 14

NORMALISATION (V. Imp)

- Concept of table design
- Which table to create, structure, columns, datatypes, widths, and constraints
- Based on User/Client requirements
- Part of design phase (minimum 1/6)
- Primary aim of normalisation is
  - To have an "efficient" table structure,
  - To avoid Data Redundancy (avoid the unnecessary duplication of data)
- Secondary aim of normalisation is to reduce the problem of insert, update and delete
- Normalisation is done from an input perspective
- Normalisation is done from a Forms perspective

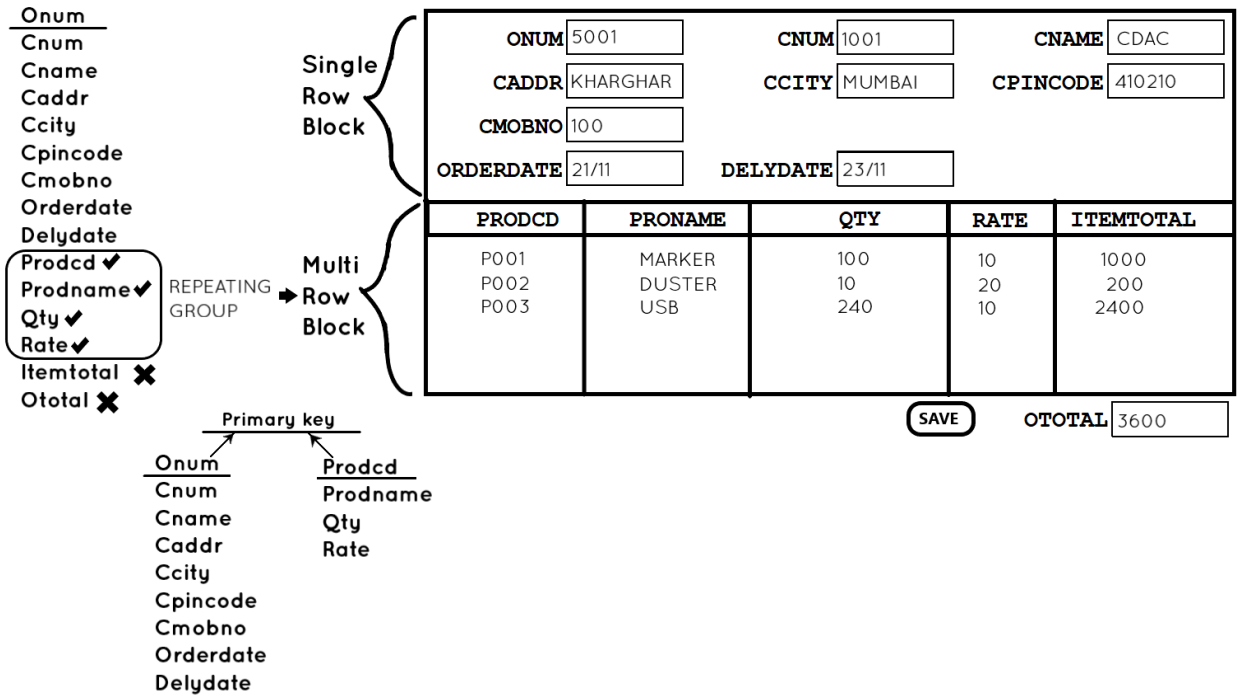
1st to 4th Normal Form → applicable for RDMS, e.g., MySQL
1st to 9th Normal Form (5th to 9th) → applicable for ORDBMS, e.g., Oracle
RDBMS + ORDBMS → ORDBMS (Object Relational DBMS)

- VIEW THE ENTIRE APPLICATION ON PER-TRANSACTION BASIS, AND YOU NORMALISE EACH TRANSACTION SEPARATELY
  e.g.,

  CUSTOMER_PLACES_AN_ORDER, CUSTOMER_CANCELS_THE_ORDER, CUSTOMER_MAKES_PAYMENT, GOODS_ARE_DELIVERED, CUSTOMER_RETURNS_THE_GOODS, etc.

Getting ready for Normalisation

e.g., CUSTOMER_PLACES_AN_ORDER



Onum	
Cnum	
Cname	
Caddr	Alphanumeric, var length, max 50 characters, etc.
Ccity	
Cpincode	Numeric, No decimal, Positive, 6 Digits, fixed length, compulsory, etc.
Cmobno	
Orderdate	
Delydate	
Prodcd	
Prodname	
Qty	
Rate	
Itemtotal	
Ototal	

1. For a Transaction, make a list of fields
2. Ask the Client for sample data
3. For every field. With inputs from the user, make a list of field properties
4. With the permission and involvement of Client, strive for Atomicity(column is made up of sub-columns, and sub-columns are made up of sub-sub-columns)
5. Get the Client sign-off
6. End of Client interaction
7. For every column assign the datatype
8. For every column, decide the width, e.g., 50
9. Assign the not null, unique and check constraints
10. For all practical purposes, you can have a single table with all these columns
11. It's recommended that every table should have a Primary key
12. Key element will be the Primary key of this table

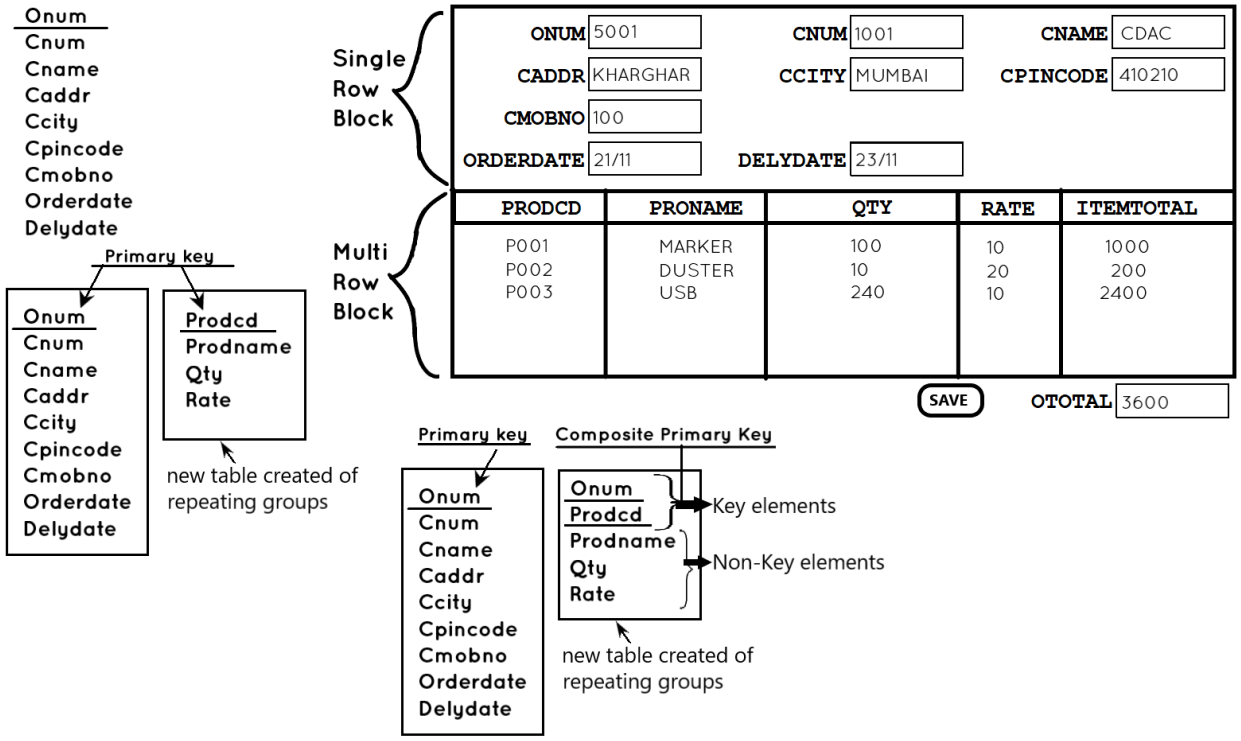
- AT THIS POINT DATA IS UN-NORMALISED FORM (UNF)
  UN-NORMALISED FORM (UNF) → Starting point of Normalisation

Optional step: -
itemtotal = qty*rate
ototal = sum(qty*rate)

- Remove the computed columns

Steps for Normalisation

1. Remove the repeating group into the new table
   Repeating group: - group of columns that constitute multi row block is known as repeating groups



2. Key element will be Primary key of new table
3. (This step may or may not be required)
   Add the Primary key of the original table to the new table to give you a Composite Primary key

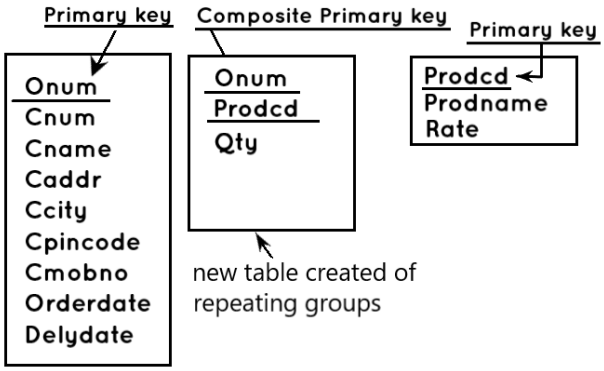
- Above 3 steps are to be repeated again and again infinitely till you cannot Normalise any further

FIRST NORMAL FORM (FNF) (Single Normal Form) (1 NF) →

Repeating groups are removed from table design

- One : Many relationship is always encountered in First Normal Form
- Department(DEPT) and Employees(EMP) tables are in First Normal Form because 1 DEPT has multiple EMPLOYEES and 1 : Many Relationship is always encountered in First Normal Form
- 25%

4. Only the tables with Composite Primary key are examined
5. Those non-key elements that are not dependent on the entire Composite Primary key, they are to be removed into a new table
6. Key elements on which originally dependent, it is to be added to the new table, and it will be the Primary key of new table



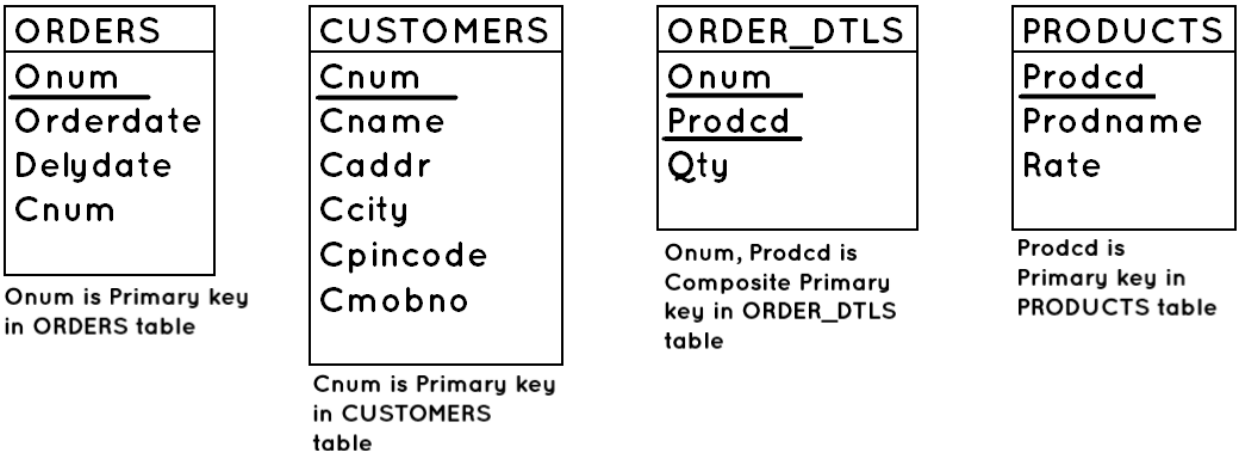
- Above 3 steps(4-5-6) are to be repeated again and again infinitely till you cannot Normalise any further

SECOND NORMAL FORM (SNF) (Double Normal Form) (2 NF) →
Every column is Functionally dependent on a Primary key
FUNCTIONAL DEPENDENCY → Every column is Functionally dependent on a Primary key means without a Primary key that column cannot function

- 67% (2/3rd)

25% + 67% = 92%
Balance 8%

7. Only the non-key elements are examined
8. Inter-dependent columns that are not dependent on Primary key, they are to be removed into a new table
9. Key element will be the Primary key of new table, and the Primary Key of new table is to be retained in the original table for Relationship purposes

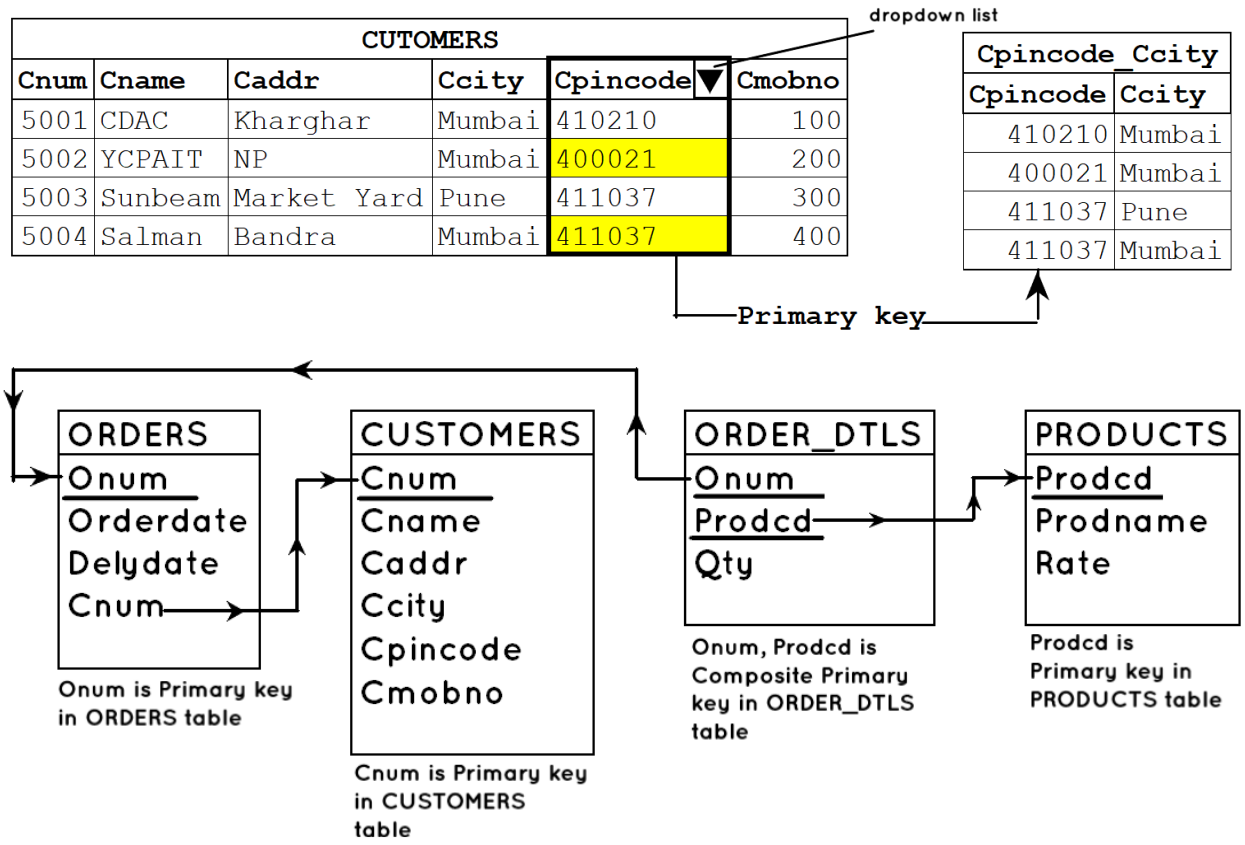


- Above 3 steps(7-8-9) are to be repeated again and again infinitely till you cannot Normalise any further

THIRD NORMAL FORM (TNF) (Triple Normal Form) (3 NF) →
Transitive dependencies(inter-dependencies) are removed from table design

FOURTH NORMAL FORM

- Extension to Third Normal Form (BCNF)
- Also known as Boyce-Codd Normal Form (BCNF)
- You may or may not implement Fourth Normal Form
- Used to protect the integrity of data
- Normally used on public network, e.g., Internet



DE-NORMALISATION

- If the data is large, if the SELECT statement are slow, then you add an extra column to the table, to improve the performance, to make the SELECT statement work faster
- Normally done for computed columns, expressions, function-based columns, summary columns, formula columns, etc.
  e.g.,
  itemtotal = qty*rate
  ototal = sum(qty*rate)
- In some of the situations, you may want to create an extra table and store the totals there
  e.g., DEPTOT table

```
+--------+--------+
| DEPTNO | SALTOT |
+--------+--------+
|      1 |  15000 |
|      2 |  6000  |
+--------+--------+
```