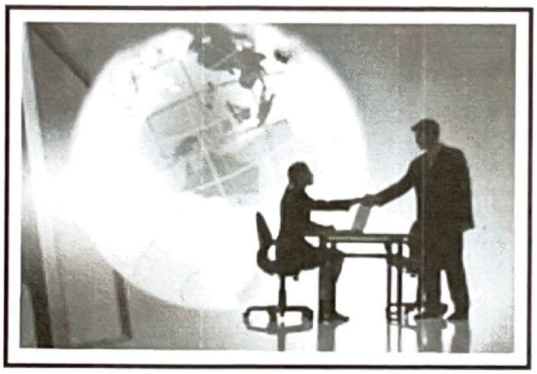
D4

SUNBEAM

SUNBEAM

Institute of Information Technology



Database Technologies Notes

Course _____ Batch _____

Student Name _____ Reg ID _____

File

- Collection of data/info on secondary storage device.
- File = Contents (Data) + Information (Metadata).
- Metadata - File Control Block (i-node)
  - Type, size, user, group, permissions, time-stamps
  - Location of data on disk.
- Files are stored in FileSystem.
- FileSystem is on disk partition, logically divided into 4 sections
  - Boot block, Super block, i-Node list, Data blocks.

DBMS - Database Management System

- Is a program that store data, retrieve data & manipulate data.
- CRUD operations
  - Create -- Insert new record.
  - Retrieve -- Select existing records.
  - Update -- Modify existing record.
  - Delete -- Delete existing record.
- Generic program for any data e.g. students, financial, employees, exams, ...
- Example: Excel/Spreadsheets, Foxpro, Dbase, ...

RDBMS - Relational DBMS

- Is a generic program that store data, retrieve data & manipulate data i.e. perform CRUD operations efficiently?
- RDBMS organize data into **Tables**, **Rows** & **Columns**. These tables are **related** to each other -- Relational DBMS.

DBMS	RDBMS
Fields, Records, File	Columns, Rows, Tables
Relations in file are program	Built-in relational feature

DBMS	RDBMS
Heavy programming	Built-in CRUD & other operations
Heavy network use (whole file)	Less network use (only record)
Client side processing	Server side processing
Slower	Faster
Huge data is not supported(MB)	Huge data support (100s of GB)
No networking support	Client-server arch
Single-user systems	Multi-user systems
File locking	Row level locking
No distribted db support	Built-in support for clustering
No security	Db auth, object level security

Different RDBMS

- MySQL: Michal - Sun - Oracle - Open Source (Community Ed).
  - Most used open source db.
  - MySQL 5.x -- MySQL 8.0.15
- MariaDb: Fully open source (clone of MySQL).
- Oracle: Enterprise database -- Express edition (educational).
- MSSQL server: Microsoft - Enterprise database -- Only windows.
- Informix: Fastest (Need assembly language knowledge).
- Sybase: Taken by SAP.
- PostreSQL: Open Source.
- DB2: IBM - Mainframe systems.
- MSAccess, Paradox, SQLite: Small RDBMS (not all features).

SQL - Structured Query Language

- RDBMS data processing is done using SQL queries.
- Client fire query on server, server process query, produce result and send result back to client.
- ANSI standardised -- 1987, 1989, ..., 2016.
- Case-insensitive language.
  - On Linux, table names & db names are case sensitive.
- Categories:
  - DDL: Data Definition Language
    - CREATE, DROP, RENAME, ALTER
  - DML: Data Manipulation Language*
    - INSERT, UPDATE, DELETE
  - DQL: Data Query Language*
    - SELECT
  - DCL: Data Control Language
    - CREATE USER, GRANT, REVOKE
  - TCL: Transaction Control Language
    - Transaction management
    - Transaction is "set of queries executed as single unit".
    - If any query fails, effect of remaining queries is discarded.
    - START TRANSACTION, SAVEPOINT, COMMIT, ROLLBACK

Naming conventions

- Table & column names can contain alphabets, digits and some special symbols e.g. $, #, _.
- Names must start with alphabets.
- If name contains special symbol, then name must be enclosed in back-quotes e.g. `T1#`
- Names can be max 30 chars long.
- Names should be readable.

MySQL - RDBMS software

MySQL Installation - Ubuntu

```
cmd> sudo apt-get install mysql-community-server mysql-community-client
```

- This installs "MySQL server" i.e. **mysqld** and MySQL client i.e. **mysql**.
- mysqld:
  - Server. No UI.
  - Implemented in C/C++.
  - Installed on RDBMS server machine.
  - Process the data & generate result.
  - Can be accessed from any MySQL client program.
    - MySQL CLI (mysql)
    - MySQL Workbench (GUI)
    - phpmyadmin (Web based)
- mysql:
  - Command line client for MySQL

```
cmd> sudo systemctl status mysql

cmd> sudo systemctl stop mysql

cmd> sudo systemctl start mysql

cmd> sudo netstat -tlnp

cmd> mysql -V
```

- During installation of mysql server, its admin user is created with name **root**. User need to specify the password.

```
cmd> mysql -u root -p

cmd> mysql -h localhost -u root -p
        # client connect to mysql server of local machine.
        # localhost can be replaced by ip addr of different server machine.
```

Creating User & Database

- It is not recommended to use root login for creating tables & doing CRUD operations.
- Better practice is to create a separate database/schema, db user and give permission to the user on that db.
- All db objects i.e. tables, constraints, relations, procedures, triggers, functions, etc should be created under some database/schema.
- Usually one project's all tables & other objects are kept in one database/schema.

```
cmd> mysql -u root -p
CREATE DATABASE dacdb;

SHOW DATABASES;

CREATE USER dac@localhost IDENTIFIED BY 'dac';

SELECT user, host FROM mysql.user;

GRANT ALL PRIVILEGES ON dacdb.* TO dac@localhost;

FLUSH PRIVILEGES;
-- activate new permissions

EXIT;
cmd> mysql -u dac -p
SHOW DATABASES;

SELECT USER(), DATABASE();

USE dacdb;

EXIT;
cmd> mysql -u dac -p
OR
cmd> mysql -u dac -pdac
OR
cmd> mysql -u dac -p dacdb
OR
cmd> mysql -u dac -pdac dacdb
```

Using MySQL

```
USE dacdb;

CREATE TABLE contacts (id INT, name VARCHAR(40), mobile VARCHAR(12), email VARCHAR(40), age INT);

SHOW TABLES;
```

```
DESCRIBE contacts;

INSERT INTO contacts VALUES (1, 'Nilesh Ghule', '9527331338', 'nilesh@sunbeaminfo.com', 35);

INSERT INTO contacts VALUES (2, 'Nitin Kudale', '9881208115', 'nitin@sunbeaminfo.com', 40), (3,
'Prashant Lad', '9881208114', 'prashant@sunbeaminfo.com', 40);

SELECT * FROM contacts;

INSERT INTO contacts (id, name, email) VALUES (4, 'Sandeep Kulange',
'sandeepkulange@sunbeaminfo.com');

INSERT INTO contacts (id, name) VALUES (5, 'Rahul Kale');

INSERT INTO contacts VALUES (5, 'Rahul Sansudi', NULL, NULL, NULL);

SELECT * FROM contacts;

SELECT id, name, mobile FROM contacts;
-- with "root" login

CREATE DATABASE classwork;

GRANT ALL PRIVILEGES ON classwork.* TO dac@localhost;

FLUSH PRIVILEGES;

QUIT;
-- with "dac" login

SHOW DATABASES;

USE classwork;

SOURCE /path/to/classwork-db.sql;

SHOW TABLES;
```

SELECT query (DQL)

```
SHOW TABLES;

SELECT * FROM BOOKS;

SELECT name, subject, price FROM BOOKS;

SELECT name, subject, price, price * 0.05 FROM BOOKS;

DESC BOOKS;
```

Computed Columns

```
SELECT name, subject, price, price * 0.05 AS gst FROM BOOKS;
-- AS keyword is optional

SELECT name, price, price * 0.05 AS gst, price + price * 0.05 total FROM BOOKS;
-- gst & total are computed columns.
```

LIMIT clause

```
SELECT * FROM BOOKS;

SELECT * FROM BOOKS LIMIT 5;
-- fetch first 5 rows

SELECT * FROM BOOKS LIMIT 4, 5;
-- skip first 4 rows and fetch next 5 rows
```

Database layout

Logical layout

- How data is represenated?
- "database" is like a namespace/container that stores all db objects related to a project.
- It contains tables, costraints, relations, stored procedures, functions, triggers, ...
- There are some system databases e.g. mysql, performance_schema, information_schema, sys, ... They contains db internal info.
    - SELECT user, host FROM mysql.user;
- The database contains tables.
- Tables have multiple columns. Each column is associated with a data-type & zero/more constaints.
- The data in table is in multiple rows. Each row have multiple values (as per columns).

Physical layout

- How data is stored on db server hard disk?
- In MySQL, the data is stored on disk in its **data directory** i.e. /var/lib/mysql
- Each database/schema is a separate sub-directory in data dir.

- Each table in the db, is a file on disk e.g. BOOKS table in classwork db is stored in file /var/lib/mysql/classwork/BOOKS.ibd.
- Note that data is stored in binary format.
- A file is not contiguously stored on hard disk & hence all data rows are not contiguous. They are scattered in the hard disk.
- In one row, all fields are consecutive.
- When records are selected, they are selected in any order. Hence we cannot fetch first n rows.

INSERT query - DML

```
USE classwork;

DESC BOOKS;

INSERT INTO BOOKS VALUES (9001, 'Atlas Shrugged', 'Ayn Rand', 'Novell', 456.562);

INSERT INTO BOOKS VALUES (9002, 'The Fountainhead', 'Ayn Rand', 'Novell', 423.123), (9003, 'The
Alchemist', 'Paulo Coelo', 'Novell', 321.345);

CREATE TABLE BOOKS2 (id INT, name VARCHAR(50), author VARCHAR(50), subject VARCHAR(50), price
DOUBLE);

INSERT INTO BOOKS2 SELECT * FROM BOOKS;

SELECT * FROM BOOKS2;

CREATE TABLE BOOKS3 (id INT, author VARCHAR(50), name VARCHAR(50), price DOUBLE, subject
VARCHAR(50));

INSERT INTO BOOKS3 SELECT id, author, name, price, subject FROM BOOKS;

SELECT * FROM BOOKS3;

TRUNCATE BOOKS3; -- delete all records

INSERT INTO BOOKS3 (id,name,author,subject,price) SELECT * FROM BOOKS;

SELECT * FROM BOOKS3;

CREATE TABLE BOOKS4 (id INT, price DOUBLE, name VARCHAR(50));

INSERT INTO BOOKS4 SELECT id, price, name FROM BOOKS;
-- OR
INSERT INTO BOOKS4 (id, price, name) SELECT id, price, name FROM BOOKS;

SELECT * FROM BOOKS4;
```

CREATE TABLE - DDL

```
CREATE TABLE BOOKS5 AS SELECT * FROM BOOKS;
-- copy schema & data

DESC BOOKS5;

SELECT * FROM BOOKS5;

CREATE TABLE BOOKS6 AS SELECT * FROM BOOKS LIMIT 0;
-- copy only schema, not data

DESC BOOKS6;

SELECT * FROM BOOKS6;

SHOW TABLES;
```

DISTINCT values

```
SELECT * FROM EMP;

SELECT * FROM DEPT;

SELECT DISTINCT deptno FROM EMP;

SELECT DISTINCT job FROM EMP;
```

ORDER BY clause

- In db rows are scattered on disk. Hence not fetched in a fixed order.

```
SELECT * FROM EMP;

SELECT * FROM EMP ORDER BY sal;

SELECT * FROM EMP ORDER BY sal DESC;

SELECT * FROM EMP ORDER BY job ASC;

SELECT * FROM EMP ORDER BY deptno;

SELECT * FROM EMP ORDER BY deptno, sal DESC;

SELECT * FROM EMP ORDER BY deptno DESC, job, sal DESC;
```

- When query is fired from client:
  i. Db server will load the data from disk into its RAM.
  ii. Using sorting algorithm data will be sorted there (on given col).
  iii. Processed result is sent back to the client.
- More are columns to sort, more is the time taken for sorting.

```
SELECT * FROM EMP
ORDER BY sal DESC
LIMIT 1;
-- highest sal

SELECT * FROM EMP
ORDER BY sal ASC
LIMIT 2,1;
-- third lowest sal
```

Criteria - WHERE clause

Relational operators

- <, >, <=, >=, =, != or <>

```
-- SELECT cols FROM table WHERE condition;

SELECT * FROM EMP WHERE empno = 7900;

SELECT * FROM EMP WHERE sal > 2500;

SELECT ename, sal, comm FROM EMP WHERE comm > 0.0;

SELECT * FROM EMP WHERE comm = NULL; -- won't work
```

NULL related operators

- NULL values cannot be compared using above relational operators.
- There are special operators for comparting NULL values:
  - IS NULL or <=>
  - IS NOT NULL

```
SELECT * FROM EMP WHERE comm IS NULL;

SELECT * FROM EMP WHERE comm <=> NULL;
```

```
SELECT * FROM EMP WHERE comm IS NOT NULL;
```

Logical Operators

- AND, OR & NOT

```
SELECT * FROM EMP WHERE job = 'SALESMAN' AND sal <= 1500;
-- find all salesman whose sal is below 1500

SELECT * FROM EMP WHERE job = 'SALESMAN' AND comm IS NULL;
-- find all salesman who are not getting comm.
-- there are no such records

SELECT * FROM EMP WHERE sal >= 1000.00 AND sal <= 2000.00;

SELECT * FROM EMP WHERE job = 'SALESMAN' OR sal >= 2500;
-- find all emps who are either salesman or have sal >= 2500.

SELECT * FROM EMP WHERE empno = 7900 OR empno = 7788;

SELECT * FROM EMP WHERE job != 'SALESMAN' AND job != 'ANALYST';
-- find all emps who are neither salesman nor analyst.

SELECT * FROM EMP WHERE NOT (job = 'SALESMAN' OR job = 'ANALYST');
-- find all emps who are neither salesman nor analyst.
```

BETWEEN operator

```
SELECT * FROM EMP WHERE sal BETWEEN 1100 AND 1600;
-- find employee with sal greater than or equal 1100 & less than or equal 1600.
-- BETWEEN includes both ends of given range.

SELECT * FROM EMP WHERE hire BETWEEN '1982-01-01' AND '1982-12-31';
-- find employee joined in year 1982.
-- dates are in format 'yyyy-mm-dd'.

SELECT * FROM EMP ORDER BY ename;

SELECT * FROM EMP WHERE ename BETWEEN 'B' AND 'F';
-- find employees whose name start from B to F.
-- FORD is excluded (due to alphabetical order).

SELECT * FROM EMP WHERE ename BETWEEN 'B' AND 'G';
SELECT * FROM EMP WHERE ename BETWEEN 'B' AND 'G' AND ename != 'G';
```

IN operator

```
SELECT * FROM EMP WHERE empno = 7900 OR empno = 7782 OR empno = 7839;
-- find employees whose empno is 7900, 7782 or 7839.

SELECT * FROM EMP WHERE empno IN (7900, 7782, 7839);
-- find employees whose empno is 7900, 7782 or 7839.

SELECT * FROM EMP WHERE job IN ('SALESMAN', 'CLERK', 'PRESIDENT');
-- find all salesman, clerk & president.

SELECT * FROM EMP WHERE deptno NOT IN (10, 20);
-- find all emps not in dept 10 & 20.
```

WHERE clause execution

- The condition in WHERE clause is checked for each row in table.
- Example:

```
SELECT * FROM EMP WHERE deptno = 30 AND sal > 1500;
```

- This query is executed similar to following C code:

```
#include <stdio.h>
struct emp {
        int empno;
        float sal;
        int deptno;
        // ...
};
int main()
{
        struct emp arr[] = { ... };
        int i;
        for(i=0; i <= 13; i++)
        {
                if(arr[i].deptno == 30 && arr[i].sal > 1500)
                {
                        printf("%d, %d, %f, ...\n", arr[i].empno, arr[i].deptno, arr[i].sal);
                }
        }
        return 0;
}
```

LIKE operator

- % is any number of any characters.
- _ is any one character.

```
SELECT * FROM EMP WHERE ename LIKE 'S%';
-- find emps whose name start with "S"

SELECT * FROM EMP WHERE ename LIKE '%ER';
-- find emps whose name end with "ER"

SELECT * FROM EMP WHERE ename LIKE '%U%';
-- find emps whose name contains "U".

SELECT * FROM EMP WHERE ename LIKE '____';
-- find emps whose names are of 4 chars.

SELECT * FROM EMP WHERE ename LIKE '%O__';
-- find emps whose name contains "O" & then two chars.
```

CASE-WHEN statement

- Used to generate a computed column in SELECT.

```
SELECT ename, deptno FROM EMP;

SELECT deptno, dname FROM DEPT;

-- 10 - ACCOUNTING, 20 - RESEARCH, 30 - SALES, 40 - OPERATIONS
SELECT ename, deptno,
CASE
WHEN deptno = 10 THEN 'ACCOUNTING'
WHEN deptno = 20 THEN 'RESEARCH'
WHEN deptno = 30 THEN 'SALES'
ELSE 'OPERATIONS'
END
FROM EMP;

SELECT ename, deptno,
CASE
WHEN deptno = 10 THEN 'ACCOUNTING'
WHEN deptno = 20 THEN 'RESEARCH'
WHEN deptno = 30 THEN 'SALES'
ELSE 'OPERATIONS'
END AS dname
FROM EMP;
```

UPDATE query - DML

- To change one or more rows in a table.
- If new value of a field needs same space as of old value, then record is updated in place (on disk).
- If size of new value is more than size of old value, then record cannot be updated in place (because it will increase the size of whole record). In this case, rdbms server create a new copy of the record with additional size requirement & then update that record. The record space is marked as blank. Such updates increase fragmentation.

```
SELECT * FROM BOOKS;

-- syntax:
-- UPDATE table SET col1=value1, ... WHERE condition;

UPDATE BOOKS SET price=150.123 WHERE id = 1004;
-- change price of book 1004 to 150.123.

UPDATE BOOKS SET price=160.123, author='K & R', name='C Programming Language' WHERE id = 1004;
-- change price of book 1004 to 160.123, author to "K & R" and name to "C Programming Language".

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

UPDATE BOOKS SET price=150.123;
-- change all books price to 150.123;

UPDATE BOOKS SET id=1005, name='C Lang', author='Ritchi', subject='C', price=123 WHERE id = 1004;
-- change all columns -- not used commonly.
```

DELETE query - DML

- One or more rows of a table are deleted.

```
DELETE FROM BOOKS WHERE id=1005;
-- delete one row

DELETE FROM BOOKS WHERE subject='Java Programming';
-- delete multiple rows

DELETE FROM BOOKS;
-- delete all rows
```

TRUNCATE query -- DDL

```
TRUNCATE TABLE BOOKS2;
```

DELETE	TRUNCATE
Table structure is not changed	Table structure is not changed
Delete all rows (if no condition)	Delete all rows
Can have WHERE clause	No WHERE clause
DML query	DDL query
Can be rollbacked (using tx)	Cannot be rollbacked
File size does not change	File size shrink
Slower	Faster

DROP query -- DDL

```
DROP TABLE BOOKS3;

DROP TABLE BOOKS7; -- error if table not present

DROP TABLE IF EXISTS BOOKS7;
```

DROP	TRUNCATE
Table struct + data is deleted	Only data is deleted
DDL Query	DDL Query
Table file is deleted.	Table file size is shrinked

Data Types

- Various data types are supported in MySQL.
- Different db support different data types (they may be similar, but not same).
- Categories:
    - Numeric types

- Integer types
  - TINYINT (1 byte)
  - SMALLINT (2 bytes)
  - MEDIUMINT (3 bytes)
  - INT (4 bytes)
  - BIGINT (8 bytes)
  - integer types can signed (default) or unsigned.
  - BIT -- number of bits can be given.
- Floating point types
  - Approx precision
    - FLOAT (4 bytes) - single precision
    - DOUBLE (8 bytes) - double precision
  - Exact precision
    - DECIMAL - size depend on given precision
      - (m,n): m digits & n digits after decimal pt

- String types
  - Char-wise storage.
    - Each char can be 1 bytes (ASCII), 2 bytes (UNICODE) & so on ... depending on char encoding.
  - CHAR(n) - n can be max 255.
    - Fixed length.
    - If given small string, rest of space is not used.
    - If given bigger string, gives error.
    - Very fast access.
  - VARCHAR(n) - n can be max 65535 (64K).
    - Variable length.
    - Internally stores chars + length.
    - Depending on entered string, space is allocated.
    - If given small string, will take less space.
    - If given bigger string, gives error.
  - TEXT
    - CHAR & VARCHAR are stored inline to the row.
    - TEXT are stored outside the record (its addr is in record).
    - Used for huge text data.
    - Very slow.

        - Different sizes:
            - TINYTEXT (max 255 chars)
            - TEXT (max 64K chars)
            - MEDIUMTEXT (max 16M chars)
            - LONGTEXT (max 4G chars)
    - Binary types
        - CHAR -- BINARY
        - VARCHAR -- VARBINARY
        - BLOB -- Similar to TEXT types
            - Stored external to record.
            - Byte-wise storage.
        - Used for binary files e.g. images, audio, video, pdf, ...
        - Using binary types increase db size quickly & slow down processing. Hence usually not recommended.
            - Different sizes:
                - TINYBLOB (max 255 bytes)
                - BLOB (max 64K bytes)
                - MEDIUMBLOB (max 16M bytes)
                - LONGBLOB (max 4G bytes)
    - Miscellaneous types
        - BOOL - Like TINYINT i.e. 1 byte.
            - TRUE - 1
            - FALSE - 0
        - ENUM - Internally an int
            - Restrict values
            - e.g. Gender: Male, Female.
            - e.g. OrderStatus: Pending, Processing, Delived, Dispatched.
    - Date/Time types
        - DATE - 3 bytes - num of days from 1-1-1000.
        - TIME - 3 bytes - timespan.
        - DATETIME - 8 bytes - store both date & time
        - TIMESTAMP - 4 bytes - num of secs from 1-1-1970
            - auto updated on DML operation.
        - YEAR - 1 byte - year 1901 to 2155

```
CREATE TABLE t1 (c1 INT, c2 INT UNSIGNED);

CREATE TABLE t2 (c1 INT, c2 INT(5), c3 INT(5) ZEROFILL);
-- size of c1, c2 & c3 is same i.e. 4 bytes.
-- (5) represent expected max num of digits.

INSERT INTO t2 VALUES (12, 12, 12);
-- c3 will be displayed as "00012".
INSERT INTO t2 VALUES (12345, 12345, 12345);
INSERT INTO t2 VALUES (1234567, 1234567, 1234567);
-- no error, accepted by db

SELECT * FROM t2;
CREATE TABLE t3 (c1 FLOAT, c2 DOUBLE, c3 DECIMAL(8,3));

INSERT INTO t3 VALUES (1.2345, 1.2345, 1.2345);

SELECT c1, c2, c3 FROM t3;

SELECT c1/11.0, c2/11.0, c3/11.0 FROM t3;
CREATE TABLE t4 (c1 CHAR(10), c2 VARCHAR(10), c3 TEXT(10));

INSERT INTO t4 VALUES ('abc', 'abc', 'abc');

INSERT INTO t4 VALUES ('abcdefghij', 'abcdefghij', 'abcdefghij');

INSERT INTO t4 VALUES ('abcdefghijk', 'abcdefghijk', 'abcdefghijk'); -- error

INSERT INTO t4 VALUES ('abcdefghij', 'abcdefghij', 'abcdefghijk');

CREATE TABLE t5 (name VARCHAR(20), birth DATETIME, modified TIMESTAMP);

DESCRIBE t5;

INSERT INTO t5 (name, birth) VALUES ('Nilesh', '1983-09-28');
INSERT INTO t5 (name, birth) VALUES ('Vishal', '1980-01-01');

SELECT * FROM t5;

INSERT INTO t5 (name, birth, modified) VALUES ('Rahul', '1980-12-31', NOW());
```

DUAL table

- A dummy/in-memory a table having single row & single column.
- It is used for arbitrary calculations, testing functions, ...

```
SELECT NOW() FROM DUAL;
```

```
SELECT 2 + 3 * 4 FROM DUAL;

SELECT USER(), DATABASE() FROM DUAL;
```

- Historically this table is first time is used by Oracle. At that time they develop table with two rows & hence name given was DUAL. Later on table changed to single row, but name kept same.
- This keyword is accepted by ANSI. But lot of db (including MySQL) keep it optional.

```
SELECT NOW();

SELECT 2 + 3 * 4;

SELECT USER(), DATABASE();
```

Help

- MySQL client provide help on commands & functions.

```
-- syntax: HELP topic;

HELP INSERT;

HELP SELECT;

HELP Functions;

HELP String Functions;
```

Single Row Functions

- These functions work on each row of table.
- **Input to fn is data from single row** & then output is single row. Hence these functions are called as "Single Row Function".

String Functions

```
HELP CONCAT;

SELECT CONCAT('Sunbeam', 'Infotech');

SELECT CONCAT('Sunbeam', ' ', 'Infotech');
```

```
SELECT ename, job FROM EMP;

SELECT CONCAT(ename, ' - ', job) AS empjob FROM EMP;
-- print "name of emp - job".

SELECT name, author FROM BOOKS;

SELECT UPPER(name), LOWER(author) FROM BOOKS;

HELP SUBSTRING;

SELECT SUBSTRING('SunBeam Infotech', 4);

SELECT SUBSTRING('SunBeam Infotech', 4, 2);

SELECT SUBSTRING('SunBeam Infotech', 4, 0); -- blank

SELECT SUBSTRING('SunBeam Infotech', 4, -2); -- blank

SELECT SUBSTRING('SunBeam Infotech', -4);

SELECT SUBSTRING('SunBeam Infotech', -8, 4);

SELECT name, SUBSTRING(author, 1, 4) FROM BOOKS;
-- print name & first 4 letters of author name

SELECT UPPER(SUBSTRING(name, -8)) FROM BOOKS;
-- print last 8 letters of name in captical case.

HELP TRIM;
-- removes leading & trailing white-spaces from string.

SELECT TRIM('                abc                 ');

SELECT * FROM BOOKS;

SELECT REPLACE(name, 'Complete', 'Full') FROM BOOKS;

SELECT ASCII('A'), ASCII('a'), CHAR(90), CHAR(122);

SELECT name, SUBSTRING(name, 3, 1) FROM BOOKS;
SELECT name, ASCII(SUBSTRING(name, 3, 1)) FROM BOOKS;
SELECT * FROM BOOKS WHERE ASCII(SUBSTRING(name, 3, 1)) BETWEEN 65 AND 90;
-- find books whose third letter is in capital.

SELECT ename, INSTR(ename, 'A') FROM EMP;

SELECT ename, INSTR(ename, 'A'), SUBSTRING(ename, INSTR(ename, 'A')+1) FROM EMP;

SELECT ename, INSTR(ename, 'A'), INSTR(SUBSTRING(ename, INSTR(ename, 'A')+1), 'A') FROM EMP;

SELECT ename FROM EMP WHERE INSTR(SUBSTRING(ename, INSTR(ename, 'A')+1), 'A') > 0;
```

```
SELECT ename FROM EMP WHERE ename LIKE '%A%';

SELECT ename, INSTR(SUBSTRING(ename, INSTR(ename, 'A')+1), 'A') FROM EMP WHERE INSTR(ename, 'A') >
0;

SELECT ename FROM EMP WHERE INSTR(ename, 'A') > 0 AND INSTR(SUBSTRING(ename, INSTR(ename, 'A')+1),
'A') = 0;
```

Numeric Functions

```
HELP Numeric Functions;

SELECT POWER(2, 10);

SELECT ABS(101.123), ABS(-101.123);

SELECT FLOOR(2.4), CEIL(2.4), FLOOR(-2.4), CEIL(-2.4);

SELECT ROUND(5432.23456, 3), TRUNCATE(5432.23456, 3);

SELECT ROUND(5432.23456, 0), TRUNCATE(5432.23456, 0);

SELECT ROUND(4567.23456, -3), TRUNCATE(4567.23456, -3);

SELECT SIGN(123), SIGN(0), SIGN(-123);

SELECT name, price, price - 500 FROM BOOKS;
SELECT name, price, SIGN(price - 500) FROM BOOKS;
SELECT name, price,
CASE
WHEN SIGN(price - 500) = 1 THEN 'High Cost'
WHEN SIGN(price - 500) = -1 THEN 'Low Cost'
END AS remark
FROM BOOKS;
```

Date Functions

```
HELP Date and Time Functions;

SELECT NOW(), SLEEP(5), SYSDATE();

SELECT DATE(NOW()), TIME(NOW());

SELECT hire, DAY(hire), MONTH(hire), YEAR(hire) FROM EMP LIMIT 3;

HELP DATE_FORMAT;

SELECT hire,
DATE_FORMAT(hire, '%d-%m-%Y') d1,
```

```
DATE_FORMAT(hire, '%d-%M-%Y') d2,
DATE_FORMAT(hire, '%d-%b-%Y') d3
FROM EMP LIMIT 3;

SELECT ename, hire,
DATEDIFF(NOW(), hire) `exp in days`,
TIMESTAMPDIFF(MONTH, hire, NOW()) `exp in months`,
TIMESTAMPDIFF(YEAR, hire, NOW()) `exp in years`
FROM EMP LIMIT 3;

SELECT ename, hire FROM EMP
WHERE YEAR(hire) IN (1981, 1983);

SELECT hire, LASTDAY(hire) FROM EMP LIMIT 3;
```

NULL Functions

- NULL represent absence of value.
- NULL is kept in record when value of Not Applicable or Not Available.
- NULL is not zero or blank string.
- When NULL value is used with most of operators & built-in functions, result is NULL.
- IS NULL, <=>, IS NOT NULL are operators for NULL value.
- ISNULL()
    - returns 1 if value is NULL.
    - similar to IS NULL operator.

```
SELECT ename, sal, comm, ISNULL(comm) FROM EMP;
```

- IFNULL()
    - IFNULL(col, value): Returns value if col is NULL.

```
SELECT ename, sal, comm, IFNULL(comm, 0) FROM EMP;

SELECT ename, sal, comm, sal + comm, sal + IFNULL(comm,0) FROM EMP;
```

- NULLIF()
    - NULLIF(val1, val2): if values are same, returns NULL.
    - If two values are diff, then return first value.

```
SELECT NULLIF('A', 'B');

SELECT NULLIF('A', 'A');

SELECT NULLIF(NULL, 'B');
```

- COALESCE()
  - COALESCE(v1, v2, v3, ...): Returns first non-null value.

```
SELECT COALESCE('A', 'B', 'C');

SELECT COALESCE(NULL, NULL, 'C');

SELECT COALESCE(NULL, NULL, NULL);
```

List Functions

- Work with list of values
- Examples: CONCAT(), COALESCE(), GREATEST(), LEAST().
- GREATEST(), LEAST() can be used with numbers, strings and dates.

```
SELECT GREATEST(34, 87, 12, 76, 43);

SELECT LEAST(34, 87, 12, 76, 43);

SELECT ename, sal, GREATEST(sal, 1000) FROM EMP;
-- display name & sal of emp. If sal below 1000, show it as 1000.
```

Aggregate Functions

- These functions work on group of rows of table.
- **Input to fn is data from multiple rows** & then output is single row. Hence these functions are called as "Multi Row Function".
- These fns are also called as "Group Functions".
- These fns are used to perform aggregate ops like sum, avg, max, min, count or std dev, etc. Hence these fns are also called as "Aggregate Functions".
- Example: SUM(), AVG(), MAX(), MIN(), COUNT().
- NULL values are ignored by group fns.

```
SELECT COUNT(sal), COUNT(comm) FROM EMP;

SELECT SUM(sal), SUM(comm) FROM EMP;
```

```
SELECT AVG(sal), AVG(comm) FROM EMP;

SELECT MAX(sal), MAX(comm) FROM EMP;

SELECT MIN(sal), MIN(comm) FROM EMP;

SELECT SUM(sal) + SUM(comm) FROM EMP;
SELECT SUM(sal + IFNULL(comm, 0)) FROM EMP;
-- find total income of all employees
```

- Cannot select group fn along with a column.
- Cannot select group fn along with a single row fn.
- Cannot use group fn in WHERE clause/condition.
- Cannot nest a grop fn in another group fn.

```
SELECT ename, SUM(sal) FROM EMP; -- error

SELECT LOWER(ename), SUM(sal) FROM EMP; -- error

SELECT * FROM EMP WHERE sal > AVG(sal); -- error

SELECT MAX(SUM(sal)) FROM EMP; -- error
```

GROUP BY clause

```
-- syntax:
-- SELECT group_col, agg_fn, ... FROM table GROUP BY group_col;

SELECT deptno, COUNT(empno) FROM EMP
GROUP BY deptno;

SELECT deptno, SUM(sal) FROM EMP
GROUP BY deptno;

SELECT deptno, COUNT(empno), SUM(sal), MAX(sal), MIN(sal), AVG(sal) FROM EMP
GROUP BY deptno;
-- find num of emps, total sal, max sal, min sal, avg sal for each dept.

SELECT DISTINCT job FROM EMP;

SELECT job, SUM(sal), AVG(sal) FROM EMP
GROUP BY job;

SELECT DISTINCT YEAR(hire) FROM EMP;

SELECT YEAR(hire), COUNT(empno) FROM EMP
GROUP BY YEAR(hire);
-- number of emp joined in each year.
```

```
SELECT deptno, job, empno FROM EMP ORDER BY deptno, job;

SELECT deptno, job, COUNT(empno) FROM EMP
GROUP BY deptno, job;

SELECT deptno, job, COUNT(empno) FROM EMP
GROUP BY deptno, job
ORDER BY deptno, job;
-- find num of emps in each dept for each job.

SELECT deptno, COUNT(empno) FROM EMP
GROUP BY deptno
ORDER BY deptno;
```

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

HAVING clause

- HAVING clause cannot be used without GROUP BY clause.
- HAVING clause is used to specify condition on aggregate values.

```
SELECT deptno, SUM(sal) FROM EMP
GROUP BY deptno;

SELECT deptno, SUM(sal) FROM EMP
GROUP BY deptno
HAVING SUM(sal) > 9000;
-- find depts which are spending more than 9000 on emp sals.
```

```
SELECT job, MAX(sal) FROM EMP
GROUP BY job
HAVING MAX(sal) > 2500;
-- find jobs which are giving max sal more than 2500.
```

- Syntactical Characteristics:
  - WHERE clause executed for each record; while HAVING is executed for each group.
  - HAVING clause can be used to specify condition on group fn or grouped columns.
  - However using HAVING to specify condition of group col reduce the performance. Use WHERE clause for the same.

```
SELECT deptno, SUM(sal) FROM EMP
GROUP BY deptno
HAVING deptno = 20;
-- find total sal of deptno 20 (unefficient)

SELECT deptno, SUM(sal) FROM EMP
WHERE deptno = 20
GROUP BY deptno;
-- find total sal of deptno 20 (efficient)
```

- SELECT Syntax

```
SELECT expr1, expr2, ... FROM table
WHERE condition
GROUP BY expr1, ... HAVING condition
ORDER BY expr1, ...
LIMIT m,n;
SELECT deptno, SUM(sal) FROM EMP
GROUP BY deptno
ORDER BY SUM(sal) DESC
LIMIT 1;
-- find dept that spend max on emp's sals.

SELECT deptno, SUM(sal) FROM EMP
WHERE job='CLERK'
GROUP BY deptno
ORDER BY SUM(sal) DESC
LIMIT 1;
-- find dept that spend max on clerk's sals.

SELECT deptno, SUM(sal) FROM EMP
WHERE job = 'CLERK'
GROUP BY deptno
HAVING SUM(sal) > 1500;
-- find dept that spend more than 1500 on clerk's sals.
```

Joins

```
USE test;

CREATE TABLE EMP(empno INT, ename VARCHAR(20), deptno INT);
INSERT INTO EMP VALUES (1, 'Sarang', 100);
INSERT INTO EMP VALUES (2, 'Nitin', 100);
INSERT INTO EMP VALUES (3, 'Amit', 200);
INSERT INTO EMP VALUES (4, 'Rahul', 200);
INSERT INTO EMP VALUES (5, 'Nilesh', 300);

CREATE TABLE DEPT(deptno INT, dname VARCHAR(20));
INSERT INTO DEPT VALUES (200, 'DEV');
INSERT INTO DEPT VALUES (300, 'QA');
INSERT INTO DEPT VALUES (400, 'ADMIN');
INSERT INTO DEPT VALUES (500, 'OPS');
| empno | ename | deptno |
| 1 | Sarang | 100 |
| 2 | Nitin | 100 |
| 3 | Amit | 200 |
| 4 | Rahul | 200 |
| 5 | Nilesh | 300 |


| deptno | dname |
| 200 | DEV |
| 300 | QA |
| 400 | ADMIN |
| 500 | OPS |

struct emp { empno, ename, deptno };
struct dept { deptno, dname };

struct emp emps[] = { ... };
struct dept depts[] = { ... };
```

Cross Join

- Join in two different tables.
- Join without any condition.
- Output rows = Num of rows in table1 * Num of rows in table2
- For each row of table1, all rows of table2 are scanned & printed.

SELECT e.ename, d.dname FROM EMP e CROSS JOIN DEPT d;

```
Q. display emp name & all possible depts he can be.
```

```
foreach(e in emps)
{
        for(d in depts)
        {
                print e.ename, d.dname;
        }
}
```

ename	dname
Sarang	DEV
Sarang	QA
Sarang	ADMIN
Sarang	OPS
Nitin	DEV
Nitin	QA
Nitin	ADMIN
Nitin	OPS
....	....

Inner Join

SELECT e.ename, d.dname FROM EMP e INNER JOIN DEPT d ON e.deptno = d.deptno;

```
Q. display emp name & dept name. Do not display non-macthing rows.
foreach(e in emps)
{
        for(d in depts)
        {
                if(e.deptno == d.deptno)
                        print e.ename, d.dname;
        }
}
```

ename	dname
Amit	DEV
Rahul	DEV
Nilesh	QA

Left Outer Join

SELECT e.ename, d.dname FROM EMP e LEFT OUTER JOIN DEPT d ON e.deptno = d.deptno;

```
Q. display emp name & dept name. If dept not found, display NULL.
foreach(e in emps)
{
        found = 0;
        for(d in depts)
        {
                if(e.deptno == d.deptno)
                {
                        found = 1;
                        print e.ename, d.dname;
                }
        }
        if(found == 0);
                print e.ename, NULL;
}
```

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

Right Outer Join

SELECT e.ename, d.dname FROM EMP e RIGHT OUTER JOIN DEPT d ON e.deptno = d.deptno;

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

Full Outer Join

- Full Outer Join is not supported in MySQL.
- But its result can be achieved by combining Left & Right joins.

Set operators

- In MySQL results of two queries can be combined using SET operators.
- There are two set operators in MySQL: UNION, UNION ALL

UNION operator

```
(SELECT e.ename, d.dname FROM EMP e
LEFT OUTER JOIN DEPT d ON e.deptno = d.deptno)
UNION
(SELECT e.ename, d.dname FROM EMP e
RIGHT OUTER JOIN DEPT d ON e.deptno = d.deptno);
```

- Common records are displayed only once.

UNION ALL operator

```
(SELECT e.ename, d.dname FROM EMP e
LEFT OUTER JOIN DEPT d ON e.deptno = d.deptno)
UNION ALL
(SELECT e.ename, d.dname FROM EMP e
RIGHT OUTER JOIN DEPT d ON e.deptno = d.deptno);
```

- Common records are displayed duplicated.

Join on multiple tables.

```
SELECT o.onum, o.amt, o.odate, c.cname FROM ORDERS o
INNER JOIN CUSTOMERS c ON o.cnum = c.cnum;

SELECT o.onum, o.amt, o.odate, s.sname FROM ORDERS o
INNER JOIN SALESPEOPLE s ON o.snum = s.snum;

SELECT o.onum, o.amt, o.odate, c.cname, s.sname FROM ORDERS o
INNER JOIN CUSTOMERS c ON o.cnum = c.cnum
INNER JOIN SALESPEOPLE s ON o.snum = s.snum;

SELECT o.onum, o.amt, o.odate, c.cname, s.sname FROM ORDERS o
INNER JOIN CUSTOMERS c ON o.cnum = c.cnum
INNER JOIN SALESPEOPLE s ON o.snum = s.snum
WHERE o.amt > 5000;
```

Cross Join vs Inner Join

- Cross Join:
  - Two different tables.
  - Doesn't have any condition.
  - To find all possible combinations.
  - Num of output rows = Num of rows in t1 * Num of rows in t2
- Inner Join:
  - Usually two different tables.
  - Have some condition (usually t1.pk = t2.fk).
  - To find rows of t1 related to rows of t2.
    - Unrelated rows from t1 & t2 are skipped.

Inner Join vs Outer Join

- Inner Join:
  - Usually two different tables.
  - Have some condition (usually t1.pk = t2.fk).
  - To find rows of t1 related to rows of t2.
    - Unrelated rows from t1 & t2 are skipped.
- Outer Join:
  - Usually two different tables.
  - Have some condition (usually t1.pk = t2.fk).

- To find rows of t1 related to rows of t2 along with additional rows in t1 (LEFT JOIN) or in t2 (RIGHT JOIN) or both (FULL JOIN).

Joins on multiple tables

```
SELECT o.onum, s.sname, c.cname FROM SALESPEOPLE s
INNER JOIN CUSTOMERS c ON c.snum = s.snum
INNER JOIN ORDERS o ON o.snum = s.snum;
-- wrong result due to wrong relation used in this query.

SELECT o.onum, o.cnum, o.snum FROM ORDERS o;

SELECT o.onum, s.sname, c.cname FROM ORDERS o
INNER JOIN CUSTOMERS c ON o.cnum = c.cnum
INNER JOIN SALESPEOPLE s ON o.snum = s.snum;
-- correct & standard.

SELECT o.onum, s.sname, c.cname
FROM ORDERS o, CUSTOMERS c, SALESPEOPLE s
WHERE o.cnum = c.cnum AND o.snum = s.snum;
-- same result (only applicable for inner joins)
```

Self Join

- Inner join or Outer join between same table. The PK & FK both belong to the same table or columns in which relation is to be made are from same table.
- For each row of a table, all rows of the same table are scanned.
- Q. Find emp name & his manager name.
  i. Assume we have two tables
     - E (employees): empno(PK), ename, mgr(FK)
     - M (managers): empno(PK), ename
  ii. Implement Join in two tables.
  iii. Since we don't have two separate table, consider same table having two copies i.e. E & M.

```
SELECT e.ename `emp_name`, m.ename `mgr_name` FROM EMP e
INNER JOIN EMP m ON e.mgr = m.empno;

SELECT e.ename `emp_name`, m.ename `mgr_name` FROM EMP e
LEFT JOIN EMP m ON e.mgr = m.empno;
```

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

```
SELECT c1.cnum, c1.cname, c2.cnum, c2.cname, c1.city
FROM CUSTOMERS c1
INNER JOIN CUSTOMERS c2 ON c1.city = c2.city
WHERE c1.cnum != c2.cnum AND c1.cnum > c2.cnum;
```

Query Performance

- Evaluating whether query execution will be faster or not. How much time (not exact time) it will take?
- To get cost of query, based on operations performed internally by database and algorithms used by database internally.
- When a query is submitted to the db server, an execution plan is made by db server. That plan has certain cost.
- More is the cost, higher will be time taken for execution of query.
- To see the execution in MySQL, there is EXPLAIN statement.

```
-- EXPLAIN query;
--              This gives high-level info about query execution.
-- EXPLAIN FORMAT=JSON query;
--              This gives low-level info (detailed) about query execution.

EXPLAIN FORMAT=JSON
SELECT * FROM EMP WHERE empno=7900; -- 1.65

EXPLAIN FORMAT=JSON
SELECT * FROM EMP WHERE job='ANALYST'; -- 1.65

EXPLAIN FORMAT=JSON
SELECT * FROM EMP WHERE mgr=7839; -- 1.65

EXPLAIN FORMAT=JSON
SELECT e.ename, d.dname FROM EMP e
INNER JOIN DEPT d ON e.deptno = d.deptno; -- 6.50
```

- Note that the query costing will vary in MySQL versions as well as machine configuration.
- The queries can be optimized by changing optimization options of database. Changing these options will force db to use different algorithm to solve the same query.
- These optimization options will change from db to db.

Indexes

- Indexes are used to improve searching speed.

```
CREATE INDEX ix_job ON EMP(job);

EXPLAIN FORMAT=JSON
SELECT * FROM EMP WHERE job='ANALYST'; -- 0.70

CREATE INDEX ix_mgr ON EMP(mgr DESC);

EXPLAIN FORMAT=JSON
SELECT * FROM EMP WHERE mgr=7839; -- 0.80

CREATE INDEX ix_edeptno ON EMP(deptno);
CREATE INDEX ix_ddeptno ON DEPT(deptno);

EXPLAIN FORMAT=JSON
SELECT e.ename, d.dname FROM EMP e
INNER JOIN DEPT d ON e.deptno = d.deptno; -- 4.52

CREATE INDEX ix_empno ON EMP(empno ASC);

EXPLAIN FORMAT=JSON
SELECT * FROM EMP WHERE empno=7900; -- 0.35

DESCRIBE EMP;

SHOW INDEXES FROM EMP;

DROP INDEX ix_empno ON EMP;
```

- Internally indexes stored addresses of rows corresponding to each value of the column on which index is created.
- These addresses are interally maintained in B-Tree or Hash data structure in MySQL.
- In index values of indexed column are arranged asc or desc order depending on how index is created.
- If index is Unique index, it cannot contain duplicate values.

```
CREATE UNIQUE INDEX ix_empno ON EMP(empno);

DESCRIBE EMP;

INSERT INTO EMP(empno, ename) VALUES (7900, 'ANDY');
```

- Composite index is created on multiple columns.
- When searching is done on all those columns (with AND), searching will be faster.

```
CREATE INDEX ix_deptno_job ON EMP(deptno, job);

EXPLAIN FORMAT=JSON
SELECT * FROM EMP WHERE deptno=30 AND job='SALESMAN'; -- 0.90
```

- When indexes are created SELECT operations on indexed columns will be very fast.
- However on each DML operation respecting index table is also modified (along with main data table) by db server. Hence DML operations will slow down.

Constraints

- Constraints are applied on columns to ensure that correct data is entered in those columns (for each row).
- The constraints are given while creating table or can be modified later (using ALTER statement).
- Types of constraints
    - Primary Key
    - Foreign Key
    - Unique Key
    - Not Null
    - Check
- Types of declaring constraints
    - Column level
        - Primary Key, Foreign Key, Unique Key, Not Null & Check constraints can be given Column level.
        - In MySQL, Foreign Key constraints at column doesn't work.
        - In MySQL, CHECK constraint at column level doesn't work.
    - Table level
        - Primary Key, Foreign Key, Unique Key & Check constraints can be given Table level.
        - In MySQL, CHECK constraint at table level doesn't work.

Primary Key

- Unique identity of each record in the table. e.g. empno in EMP table.
- PK is always unique and cannot be null.
- PK shouldn't be changed.

(Natural) Primary Key

- A col which represent natural identity of row, should be taken as PK. e.g. Users table PK can be email.
- Internally a UNIQUE index is auto created for PK column.

```
CREATE TABLE users (
name VARCHAR(50),
email VARCHAR(50) PRIMARY KEY,
password VARCHAR(50),
age INT
);

-- OR

CREATE TABLE users (
name VARCHAR(50),
email VARCHAR(50),
password VARCHAR(50),
age INT,
PRIMARY KEY (email)
);
```

Composite Primary Key

- Sometimes a combination of multiple columns is unique (instead of single column). In this case we create composite PK.
- Internally a UNIQUE Composite index is auto created for PK columns.
- Example: STUDENTS table:

roll	std	name	...
1	1	A	
2	1	B	

roll	std	name	...
1	2	C	

```
CREATE TABLE students(
roll INT,
std INT,
name VARCHAR(50),
PRIMARY KEY (roll, std)
);
```

Surrogate Primary Key

- When no column can be used as unique identity, neither combination columns is available to use as unique identity; a separate column is added into table to use as PK.
- Such PK is called as "Surrogate Primary Key".
- Such unique nums are usually generated using sequences in oracle or auto_increment in MySQL.

```
CREATE TABLE registrations(
regno INT AUTO_INCREMENT,
name VARCHAR(50),
email VARCHAR(50),
batch INT,
PRIMARY KEY(regno)
);

INSERT INTO registrations(name, email, batch) VALUES ('A', 'a@sun.com', 100); -- regno will be 1
INSERT INTO registrations(name, email, batch) VALUES ('B', 'b@sun.com', 101); -- regno will be 2
```

Add or remove PK

- If table is already created, its PK can be added or removed.

```
USE sales;

ALTER TABLE CUSTOMERS ADD PRIMARY KEY (cnum);

ALTER TABLE ORDERS ADD PRIMARY KEY (onum);

ALTER TABLE SALESPEOPLE ADD PRIMARY KEY (snum);

DESC CUSTOMERS;
DESC ORDERS;
DESC SALESPEOPLE;
```

Deciding Primary Key

- Primary is decided when table is created.
- The table design of a project is done using a process, called as "Normalization".
- PKs is by-product of Normalization process.

Unique Key

- Cannot be duplicated.
- Can be NULL.
- There can be multiple Unique keys in a table.
- Internally UNIQUE index is created on that column.

```
CREATE TABLE users (
name VARCHAR(50),
email VARCHAR(50),
password VARCHAR(50),
age INT,
mobile CHAR(20) UNIQUE,
PRIMARY KEY (email)
);
```

NOT NULL Constraint

- The value of col cannot be NULL.

```
CREATE TABLE users (
name VARCHAR(50) NOT NULL,
email VARCHAR(50),
password VARCHAR(50) NOT NULL,
age INT,
mobile CHAR(20) UNIQUE,
PRIMARY KEY (email)
);
```

Foreign Key Constraint

- A column (or combn of column) that refer to another col into another/same table.
- Usually FK col refer to the PK col of another table.
- If corresponding PK in another table is not available, then FK entry cannot be done. It raise error.

- For example: If DEPT table contains deptnos 10, 20, 30 & 40; then a row EMP table cannot have deptno=50.
- Internally it create an index for FK column.
- Having indexes (auto created for PK & FK) improve speed of join.

```
CREATE TABLE CUSTOMERS(
cnum INT,
cname VARCHAR(20),
city VARCHAR(20),
rating INT,
snum INT,
FORIEGN KEY (snum) REFERENCES SALESPEOPLE(snum)
);
-- OR
ALTER TABLE CUSTOMERS ADD FOREIGN KEY (snum) REFERENCES SALESPEOPLE(snum);

EXPLAIN FORMAT=JSON
SELECT c.cname, s.sname FROM CUSTOMERS c
INNER JOIN SALESPEOPLE s ON c.snum = s.snum;

SELECT * FROM SALESPEOPLE;

INSERT INTO CUSTOMERS (cnum, cname, snum) VALUES (9001, 'Xyz', 1009);
-- error (no salesman in SALESPEOPLE with snum=1009).

DELETE FROM SALESPEOPLE WHERE snum = 1001;
-- error (it will make corresponding CUSTOMER records orphan).

UPDATE SALESPEOPLE SET snum = 2001 WHERE sname = 'Peel';
-- error (it will make corresponding CUSTOMER records orphan).
```

- In MySQL, if parent row is deleted/updated corresponding child rows can be deleted/updated; if it is mentioned while creating table.

```
CREATE TABLE CUSTOMERS(
cnum INT,
cname VARCHAR(20),
city VARCHAR(20),
rating INT,
snum INT,
FORIEGN KEY (snum) REFERENCES SALESPEOPLE(snum) ON DELETE CASCADE
);
-- OR
CREATE TABLE CUSTOMERS(
cnum INT,
cname VARCHAR(20),
city VARCHAR(20),
rating INT,
snum INT,
FORIEGN KEY (snum) REFERENCES SALESPEOPLE(snum) ON UPDATE CASCADE
);
```

Disabling Foreign Key checks

- We can disable foreign key checks temporarily.
- This may loose integrity of data.
- This feature can be helpful while migrating data into new database, to speed up data transfer.

```
SELECT @@foreign_key_checks;
        -- If 1, FK checks are done.
        -- If 0, FK checks are skipped.

SET @@foreign_key_checks = 0;

SELECT @@foreign_key_checks;

INSERT INTO CUSTOMERS (cnum, cname, snum) VALUES (9001, 'Xyz', 1009);
```

CHECK Constraint

- MySQL support CHECK constraint from 8.0.16 onwards.
- It is used to validate values to be inserted/updated in column.
    - e.g. sal must be greater than 500.
    - e.g. age must be in range of 18 to 58.
    - e.g. Name must be entered in cap case.
    - e.g. sal + comm should not exceed 10000.
- MySQL can emulate CHECK constraints using Triggers (BEFORE INSERT & BEFORE UPDATE).

```
SELECT @@version;

CREATE TABLE EMP(
empno INT,
ename VARCHAR(20) CHECK (ename = UPPER(ename)),
age INT CHECK (age BETWEEN 10 AND 58),
sal DOUBLE CHECK (sal > 500),
comm DOUBLE,
CHECK ((sal + comm) <= 10000)
);
```

Sub-query

- Is query inside query. Typically SELECT query within SELECT query.
- For each row of outer query, inner query is executed once.

- If db is not doing any optimization, then sub-queries are usually slower than joins.

Single row sub-query

- Sub-query returns single row.

```
-- Q. find all emps whose sal is greater than avg sal.
SELECT AVG(sal) FROM EMP;
SELECT * FROM EMP WHERE sal > 2073.21;

-- Q. find all emps whose sal is greater than avg sal (in a query).
SELECT * FROM EMP WHERE sal > (SELECT AVG(sal) FROM EMP);

-- Q. find emp with 3rd highest sal.
SELECT DISTINCT sal FROM EMP ORDER BY sal DESC; -- step1
SELECT DISTINCT sal FROM EMP ORDER BY sal DESC LIMIT 2,1; -- step2
SELECT * FROM EMP WHERE sal = 2975.00; -- answer

-- Q. find emp with 3rd highest sal (in a query).
SELECT * FROM EMP WHERE sal = (SELECT DISTINCT sal FROM EMP ORDER BY sal DESC LIMIT 2,1);

-- Q. find emps whose sal is greater than all salesman.
SELECT * FROM EMP WHERE job='SALESMAN'; -- step1
SELECT MAX(sal) FROM EMP WHERE job='SALESMAN'; -- step2
SELECT * FROM EMP WHERE sal > 1600.00; -- answer

-- Q. find emps whose sal is greater than all salesman (in a query)
SELECT * FROM EMP WHERE sal > (SELECT MAX(sal) FROM EMP WHERE job='SALESMAN');
```

Multi-row sub-query

- Sub-query returns multiple rows.
- ALL operator is used to compare (relational op) result of multi-row sub-query into outer query. It compare will all values returned by sub-query. It is equivalent to logical AND.
- ANY operator is used to compare (relational op) result of multi-row sub-query into outer query. It compare will all values returned by sub-query. It is equivalent to logical OR.

```
-- Q. find emps whose sal is greater than all salesman.
SELECT sal FROM EMP WHERE job='SALESMAN'; -- step1
SELECT * FROM EMP
WHERE sal > ALL(SELECT sal FROM EMP WHERE job='SALESMAN'); -- ans

-- Q. find emps whose sal is greater than any salesman.
SELECT * FROM EMP
WHERE sal > ANY(SELECT sal FROM EMP WHERE job='SALESMAN')
AND job != 'SALESMAN'; -- ans
```

```
-- Q. find all depts who contain at least one emp.
SELECT * FROM DEPT; -- step1
SELECT * FROM EMP; -- step2
SELECT DISTINCT deptno FROM EMP; -- step3
SELECT * FROM DEPT
WHERE deptno = ANY(SELECT DISTINCT deptno FROM EMP);

-- Q. find all depts who contain at least one emp.
SELECT * FROM DEPT
WHERE deptno IN (SELECT DISTINCT deptno FROM EMP);
```

ALL vs ANY

ALL	ANY
Used with multi-row sub-qry	Used with multi-row sub-qry
Can be used with any reln op	Can be used with any reln op
Like logical AND	Like logical OR

IN vs ANY

IN	ANY
Used with/without sub-query	Used with multi-row sub-qry
Like logical OR	Like logical OR
Can be used to check equality	Can be used with any reln op
Faster (for equality)	Slower (for equality)

Correlated Sub-query

- For each row of outer query, inner query is executed once.
- To improve the performance of sub-queries, inner query should return min number of rows.
- This can be done by giving criteria (WHERE clause) in inner query based on value of row in outer query.
- Such sub-queries where result of inner query depends on current row of outer query, is called as "Correlated Sub-query".

```
-- Q. find all depts who contain at least one emp.
SELECT * FROM DEPT
WHERE deptno IN (SELECT deptno FROM EMP); -- subqry return 14 rows

-- Q. find all depts who contain at least one emp.
-- EXPLAIN FORMAT=JSON
SELECT * FROM DEPT d
WHERE d.deptno
IN (SELECT e.deptno FROM EMP e WHERE e.deptno = d.deptno);
-- subqry returns 3 rows when d.deptno = 10, 5 rows when d.deptno = 20, 6 rows when d.deptno = 30
and 0 rows when d.deptno = 40.
-- reduce num of rows returned by subqry - to improve performance

-- Q. find all depts who contain at least one emp.
-- EXPLAIN FORMAT=JSON
SELECT * FROM DEPT d WHERE
(SELECT COUNT(e.deptno) FROM EMP e WHERE e.deptno = d.deptno) > 0;

-- Q. find all depts who contain at least one emp.
-- EXPLAIN FORMAT=JSON
SELECT * FROM DEPT d WHERE
EXISTS (SELECT e.deptno FROM EMP e WHERE e.deptno = d.deptno);

-- Q. find all depts who doesn't contain any emp.
SELECT * FROM DEPT d WHERE
(SELECT COUNT(e.deptno) FROM EMP e WHERE e.deptno = d.deptno) = 0;

-- Q. find all depts who doesn't contain any emp.
SELECT * FROM DEPT d WHERE
NOT EXISTS (SELECT e.deptno FROM EMP e WHERE e.deptno = d.deptno);
```

SELECT within UPDATE/DELETE

- You cannot delete/update from the same table, on which you are firing inner query.

```
-- Delete all emps whose sal is greater than avg sal of emps.
SELECT * FROM EMP WHERE sal > (SELECT AVG(sal) FROM EMP); -- step1

DELETE FROM EMP WHERE sal > (SELECT AVG(sal) FROM EMP); -- ans
```

- You can delete/update from different table than the table on which inner query is fired. Correlated sub-query is not allowed.

```
-- Delete all depts, who doesn't contain emp.
DELETE FROM DEPT
WHERE deptno NOT IN (SELECT deptno FROM EMP);
```

Transactions

- Set of DML queries executed as single unit. If any one query is failed, effect of all queries in tx is discarded.
- Tx very useful on real world applications e.g. Funds transfer, Placing order, Reservation, etc.
- TCL command:
  - START TRASACTION;
  - COMMIT;
  - ROLLBACK;
  - SAVEPOINT x;
  - ROLLBACK TO x;
- **START TRANSACTION;** command is used to begin a new tx.
- **COMMIT;** command is used to commit current tx. All changes done by queries with this tx are saved permanently in db.
- **ROLLBACK;** command is used to rollback current tx. All changes done by queries with this tx are discarded permanently from db.
- Once commit/rollback is done, current tx will finish. For further changes start new tx.

Savepoint

- Savepoints internally store a state within tx.
- It is used rollback to that state, if we want to discard tx partially.
- We can only rollback to savepoint. We cannot commit to savepoint.
- When commit/rollback is done for a tx, all savepoints in that tx are deleted.

```
START TRANSACTION;

-- dml q1
-- dml q2
SAVEPOINT sa;

-- dml q3
-- dml q4
SAVEPOINT sb;

-- dml q5
-- dml q6

ROLLBACK TO sa; -- discard changes of d3, q4, q5 & q6
```

```
COMMIT; -- save changes of q1 & q2
START TRANSACTION;
-- dml q1
-- dml q2
SAVEPOINT sa;
-- dml q3
-- dml q4
SAVEPOINT sb;

-- dml q5
-- dml q6

COMMIT TO sb; -- commit to savept is not allowed
ROLLBACK TO sb; -- instead rollback to savept
COMMIT; -- and then commit.
```

autocommit variable

- There is a global config variable *autocommit*.
- In MySQL, by default it is 1. i.e. all changes are auto commited.
- To begin tx, alternative way is to set *autocommit* to 0.
- As a beginner do not use this variable.

```
SELECT @@autocommit;

SET @@autocommit = 0;

DELETE FROM DEPT;

ROLLBACK;

SET @@autocommit = 1;
```

DDL queries in tx

- DDL queries are not part of tx.
- When DDL query is executed, current tx is automatically commited.
- So first commit/rollback current tx & then only fire DDL queries.

```
START TRANSACTION;

DELETE FROM DEPT;

SHOW TABLES;
```

```
DROP TABLE DUMMY;

ROLLBACK;

SELECT * FROM DEPT;
```

Transaction characteristics

- Tx follow ACID properties.
- A : Atomic
  - At the end of tx all queries in tx will succeed or will discard.
  - Partial success/discard is not allowed.
- C : Consistent
  - At the end of tx changes saved, will be visible to all clients.
  - Different clients do not see different state of data.
- I : Isolated
  - Num of clients can execute txs simultenously.
  - They should be able to execute tx without affecting each other, as if they are done one after another.
  - Internally db server maintains a req queue, that stores all ops from all clients. Then they are executed one after another.
- D : Durable
  - At the end of tx all changes must be saved permanently on db server disk.
  - Data changes should not be loss (even in case of server crash).

Transaction internals

- For each tx a separate temp table is created.
- All changes done by user in that tx are recorded in temp table, but not updated in main table.
- When user fetch records in that tx, combined result of main table & tx temp table is shown to him.
- However these changes are not visible to other users, until tx is commited by the first user.

```
-- root login

SELECT user, host FROM mysql.user;

CREATE USER dac@localhost IDENTIFIED BY 'dac';
```

```
GRANT ALL PRIVILEGES ON classwork.* TO dac@localhost;

FLUSH PRIVILEGES;

SHOW GRANTS FOR nilesh@localhost;

QUIT;
```

Changes in a tx of client are not visible to another client.

- And will be discarded, if tx is rollbacked by user1.

```
-- (terminal1) cmd> mysql -u nilesh -pnilesh classwork

ALTER TABLE DEPT ADD PRIMARY KEY(deptno); -- 0

START TRANSACTION; -- 1
SELECT * FROM DEPT; -- 2
DELETE FROM DEPT WHERE deptno=40; -- 4
ROLLBACK; -- 6
-- (terminal2) cmd> mysql -u dac -pdac classwork

SELECT * FROM DEPT; -- 3
SELECT * FROM DEPT; -- 5 (changes in 4 are not visible here)
SELECT * FROM DEPT; -- 7 (changes in 4 are discarded)
```

Changes in a tx of client are not visible to another client.

- And will be visible to user2, if tx is commited by user1.

```
-- (terminal1) cmd> mysql -u nilesh -pnilesh classwork

START TRANSACTION; -- 1
SELECT * FROM DEPT; -- 2
UPDATE DEPT SET loc='PUNE' WHERE deptno=40; -- 4
COMMIT; -- 6
-- (terminal2) cmd> mysql -u dac -pdac classwork

SELECT * FROM DEPT; -- 3
SELECT * FROM DEPT; -- 5 (changes in 4 are not visible here)
SELECT * FROM DEPT; -- 7 (changes in 4 are visible here)
```

Optimistic Locking.

```
-- (terminal1) cmd> mysql -u nilesh -pnilesh classwork

START TRANSACTION; -- 1
SELECT * FROM DEPT; -- 2
UPDATE DEPT SET loc='PUNE' WHERE deptno=40; -- 4
```

```
SELECT * FROM DEPT; -- 5
COMMIT; -- 8
-- (terminal2) cmd> mysql -u dac -pdac classwork

SELECT * FROM DEPT; -- 3
SELECT * FROM DEPT; -- 6 (changes in 4 are not visible here)
DELETE FROM DEPT WHERE deptno=40; -- 7 (blocked until 8 is done)
SELECT * FROM DEPT; -- 9 (record 40 is deleted)
```

Pessimistic Locking.

```
-- (terminal1) cmd> mysql -u nilesh -pnilesh classwork

START TRANSACTION; -- 1
SELECT * FROM DEPT WHERE deptno=30 FOR UPDATE; -- 2
UPDATE DEPT SET loc='KARAD' WHERE deptno=30; -- 4
SELECT * FROM DEPT; -- 5
COMMIT; -- 6
-- (terminal2) cmd> mysql -u dac -pdac classwork

SELECT * FROM DEPT WHERE deptno=30 FOR UPDATE; -- 3 (will block until 6)
```

Views

- View is limited/restricted **view** of the data.
- View is used for frequently required SELECT queries. SELECT queries can be with few cols, few rows, grouped, joins or sub-queries.
- It avoid need of rewriting same query multiple times.
- There is no memory/space allocated to view. Working on view internally is working underlying table(s).
- View doesn't increase speed of execution. Because it is merely executing same SELECT query.
- Two types of views
    - Simple view
    - Complex view

Simple view

- Doesn't contain computed column, group by, sub-queries or joins.
- On which we can perform SELECT as well as DML operations.
- DML operations are allowed as long as no constraint on table is violated.

- If you want to allow DML operations only for selected rows (as given in WHERE clause), then that view should created **WITH CHECK OPTION**.

```
USE sales;

CREATE VIEW v_customers
AS SELECT * FROM CUSTOMERS;

SHOW TABLES;

SELECT * FROM v_customers;

CREATE VIEW v_customer_details
AS SELECT cnum, cname, city, rating FROM CUSTOMERS;

SELECT * FROM v_customer_details;

UPDATE v_customer_details SET city='Paris' WHERE cname='Hoffman';

INSERT INTO v_customer_details VALUES (2010, 'Bill', 'Karad', 500);

CREATE VIEW v_toprated_customers
AS SELECT * FROM CUSTOMERS WHERE rating >= 200;

SELECT * FROM v_toprated_customers;

CREATE VIEW v_small_orders
AS SELECT * FROM ORDERS WHERE amt <= 1000 WITH CHECK OPTION;

SELECT * FROM v_small_orders;

INSERT INTO v_small_orders VALUES (6001, 9000, '2019-03-09', 2010, 1001);
INSERT INTO v_small_orders VALUES (6002, 8000, '2019-03-09', 2010, 1001);
```

Complex view

- Can perform only SELECT operations.
- Cannot perform DML operations.
- Contain computed column, group by, sub-queries or joins.

```
CREATE VIEW v_year_orders
AS SELECT onum, amt, odate, snum, cnum, YEAR(odate) oyear FROM ORDERS;

SELECT * FROM v_year_orders WHERE oyear=2019;

CREATE VIEW v_customer_maxamt
AS SELECT cnum, MAX(amt) FROM ORDERS GROUP BY cnum;

SELECT * FROM v_customer_maxamt;
```

```
-- INSERT INTO v_customer_maxamt VALUES (2010, 20000); -- error

SELECT * FROM v_customer_maxamt WHERE cnum IN (2001, 2002, 2003);

CREATE VIEW v_toporders
AS SELECT * FROM ORDERS
WHERE amt > (SELECT AVG(amt) FROM ORDERS);

SELECT * FROM v_toporders;

CREATE VIEW v_orders
AS SELECT o.onum, o.amt, o.odate, o.cnum, c.cname, c.rating crating, o.snum, s.sname, s.comm scomm
FROM ORDERS o
INNER JOIN CUSTOMERS c ON o.cnum = c.cnum
INNER JOIN SALESPEOPLE s ON o.snum = s.snum;

SELECT * FROM v_orders;

SELECT onum, snum, amt, scomm, scomm * amt FROM v_orders;
```

Applications

- Simplifies some complex queries.
- Enables reuse of queries
- Can provide restricted view of data to the different users.
- Hide source code of table.
- Even though table structure is changed, applications can continue to execute.

DCL - Data Control Language

- It common requirement to provide limited to access of the data to certain users.
- Limited number of rows, limited number of columns or limited privileges of table/view.
- This is done using views & DCL.
- DCL Commands
    - GRANT
    - REVOKE
    - SHOW GRANTS
- When db is installed, one user is created by default. It is admin user. In MySQL admin user login name is "root".
- By default "root" have all permissions.

Basic commands

```
SHOW DATABASES;

SHOW TABLES;

SELECT user, host FROM mysql.user;

CREATE USER username@hostname IDENTIFIED BY 'password';

SHOW GRANTS FOR username@hostname;

SELECT USER(), DATABASE();
```

Privileges

- Two types of privileges (MySQL):
  - Global privileges
    - Not specific to db or table.
    - e.g. privileges to CREATE USER, CREATE DATABASE, DROP DATABASE, changing system variables, ...
    - These privileges are given `ON *.*`.
  - Object level privileges
    - Privileges inside db/schema on table or views.
    - These privileges are given `ON dbname.*`.
- Different privileges (MySQL):
  - USAGE: No permissions
  - SELECT: DQL permissions
  - INSERT,UPDATE,DELETE: DML permssions
  - CREATE,ALTER,DROP: DDL permssions
  - ALL: all permissions except *GRANT OPTION*
  - GRANT OPTION: permission to give permissions to others.
    - You can only give those permissions to others, which you have.
  - SUPER: admin permission to change system variables.
  - EXECUTE: permission to execute stored procedure.

```
-- Syntax
GRANT permissions ON dbname.* TO user@host;

GRANT permissions ON tablename.* TO user@host;
```

```
GRANT permissions ON viewname.* TO user@host;
-- root login

-- don't do this.
GRANT ALL PRIVILEGES ON *.* TO nilesh@localhost WITH GRANT OPTION;
-- will make nilesh similar to admin.
-- strictly prohibited for security reasons.

GRANT ALL PRIVILEGES ON hr.* TO nilesh@localhost WITH GRANT OPTION;
-- will make nilesh admin for hr db.

FLUSH PRIVILEGES;

-- nilesh login
USE hr;

GRANT SELECT ON departments TO dac@localhost;

GRANT INSERT,UPDATE,DELETE,SELECT ON employees TO dac@localhost;

-- dac login

SHOW DATABASES;

USE hr;

SHOW TABLES;

SHOW GRANTS FOR dac@localhost;

CREATE TABLE temp(id INT);
-- not allowed

SELECT * FROM employees;

-- INSERT INTO employees VALUES (...); -- allowed

SELECT * FROM departments;

-- INSERT INTO departments VALUES (...); -- not allowed

-- nilesh login

USE hr;

REVOKE UPDATE, DELETE ON employees FROM dac@localhost;

-- dac login

SHOW GRANTS;
```

MySQL Variables

- There are two types of variables
  - System variables
  - User defined variabless

System variables

```
- They are pre-defined in MySQL.
- They are accessed using @@ sign.
- To see variables:
        - SHOW VARIABLES;
        - SHOW VARIABLES LIKE '%foreign%';
- Few variables can be modified in user session scope, while few can be modified globally.
- Changes in session scope are visible only to current user (not impacting other users).
        - e.g. @@autocommit, @@foreign_key_checks, @@optimizer_switch, ...;
        - SET @@autocommit = 0;
        - SELECT @@version;
- Changes in global scope are allowed only to user with SUPER privileges (like root).
        - SET GLOBAL @@foreign_key_checks = 0;
- Changes in global scope are visible all users;
```

User defined variables

- Defined by individual users.
- Only accessible in that users that session.
- Accessed using @ sign.
- e.g. SET @a = 1;
- e.g. SELECT @a;

MySQL Programming

- RDBMS Programming there is ISO standard PSM (part of SQL std).
- SQL/PSM stands for **Persistent Stored Module**.
- This std is started from year 1992 (as part of SQL std).
- Inspired from PL/SQL - Programming language of Oracle.
- PSM allows writing programs for RDBMS. The program contains set of SQL statements along with programming constructs e.g. variables, if-else, loops, case, ...

- PSM is a block language. Blocks can be nested into another block.
- Advantages of block language.
  - Modularity
  - Reusability
  - Abstraction
- MySQL program can be a stored procedure, function or trigger.
- MySQL programs are written by db users(programmers) from client and then submitted to db server. Server check syntax and store them into db in binary form (compilation).
- These programs are later executed by db users. Since programs are stored on server in compiled form, their execution is very fast.
- Note that all these programs will run in server memory.

Stored Procedure

- Stored Procedure is a **routine**. It contains multiple SQL statements along with programming constructs.
- Procedure doesn't return any value (like void fns in C).
- Procedures can take zero or more parameters.
- Procedures are createds using **CREATE PROCEDURE** and deleted using **DROP PROCEDURE**.
- Procedures are invoked/called using **CALL** statement.
- In general, procedures don't print results on console.
  - They can return results via OUT parameters.
  - They can insert results into some table.
- In MySQL, we can use SELECT statement at the end of SP to produce results (which will be displayed on console).

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

DELIMITER $$

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

DELIMITER ;

CALL sp_hello();
```

```
SELECT * FROM result;
```

- It is easier to type SP using some editor in a .sql file and then use that .sql file using SOURCE keyword.

PSM Syntax

Variables

```
DECLARE varname DATATYPE;

DECLARE varname DATATYPE DEFAULT init_value;

SET varname = new_value;

SELECT new_value INTO varname;

SELECT expr_or_col INTO varname FROM table_name;
```

Parameters

```
CREATE PROCEDURE sp_name(p1 DATATYPE)
BEGIN
-- sp definition
END;

CREATE PROCEDURE sp_name(p1 DATATYPE, p2 DATATYPE)
BEGIN
-- sp definition
END;
```

IF and IF-ELSE

```
IF condition THEN
        body;
END IF;

IF condition THEN
        if-body;
ELSE
        else-body;
END IF;

IF condition THEN
        if1-body;
ELSE
```

```
        IF condition THEN
                if2-body;
        ELSE
                else2-body;
        END IF;
END IF;

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

CASE-WHEN

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

WHILE loop

```
* Similar to "while" in C.
* Check condition at start of loop.
* Loop is repeated, if condition is true.

WHILE condition DO
        body;
END WHILE;
```

REPEAT-UNTIL loop

- Similar to "do-while" in C.
- Check condition at end of loop. Hence loop executed at least once.
- Loop is repeated, if condition is false.

```
REPEAT
        body;
UNTIL condition
END REPEAT;
```

LOOP (labeled)

- Like infinite loop (i.e. no condition given).
- You can check condition in loop and then exit the loop (using LEAVE statement).
- LEAVE statement is like "break" in C.
- ITERATE statement is like "continue" in C.
- Loop is associated with a label. It is useful to exit from outer loop, while in inner loop.

```
label: LOOP
        ...
        IF condition THEN
                ...
                LEAVE label;
        END IF;
        ...
END LOOP;
```

Parameter types

```
CREATE PROCEDURE sp_name(PARAMTYPE p1 DATATYPE, PARAMTYPE p2 DATATYPE)
BEGIN
        ...
END;
```

- In PSM, SP can have three types of params.
  - IN param: To give input to SP.
    - Initialized by calling program.
    - By default, all params are IN.
  - OUT param: To take output from SP.
    - Initialized by called SP.
  - INOUT param: To give input to SP and to take output from SP.
    - Initialized by calling program and modified by called SP.

```
void add(int a, int b, int *c)
{
        // a, b are IN params; while c is OUT param.
        *c = a + b;
}
void sqr(int *a)
{       // a is INOUT param
        *a = (*a) * (*a);
}
int main()
{
```

```
        int x=10, y=3, z, w=5;
        add(x, y, &z);
        sqr(&w);
        // ...
        return 0;
}
```

Error/Exception handling

- Exceptions are runtime problems.
- They can be handled in MySQL SQL/PSM using error handlers.
- Runtime errors:
  - 1062 - Duplicate Entry (PK).
  - 1044 - Access Denied.
- Syntax of error handler:
  - DECLARE action HANDLER FOR error handler_implementation;
- **action**
  - Two types of actions:
    - EXIT: SP will exit after execution of handler.
      - Next statements (after statement at which error occurred) will not be executed.
    - CONTINUE: SP will continue to execute after execution of handler.
      - Next statements (after statement at which error occurred) will be executed.
- **error**
  - Three ways of specifying error.
    - MySQL error code: e.g. 1062, 1044, ...
    - MySQL SQLSTATE value: e.g. 23000, 42000, NOT FOUND, SQL WARNING, ...
    - Named condition: giving alias to error code. DECLARE duplicate CONDITION FOR 1062;
- **handler_implementation**
  - Two ways of writing handler:
    - Single Liner:
      - Written on same line.
    - PL Block:
      - Written in PL block i.e. BEGIN ... END;
      -

Cursors

- Cursor is a special variable used to read records/rows from the table(s) **one by one**.
- Based on SELECT statement. It should be mentioned into cursor declaration. SELECT statement can be simple or with criteria, order, limit, group, sub-query and/or join.
- Cursor programming steps:
  i. Declare err handler for cursor (NOT FOUND err like EOF).
  ii. Declare a cursor variable & its SELECT statement.
  iii. Open cursor.
  iv. Fetch values from cursor into local variables & process them.
  v. Once rows are completed, err handler will be executed.
  vi. Close cursor.
- Cursor Syntax:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND --1
BEGIN
        SET v_flag = 1; --5
END;
DECLARE v_curname CURSOR FOR select_statement; --2
OPEN v_curname; --3
label: LOOP
        FETCH v_curname INTO variable(s); --4
        IF v_flag = 1 THEN
                LEAVE label;
        END IF;
        process variables; --4
        ...
END LOOP;
CLOSE v_curname; -- 6
```

- MySQL Cursor characteristics:
  - Readonly
    - Using cursor we can only read not modify or delete.
    - The cursor variable is not meant to be modified.
  - Non-Scrollable
    - Cursor is forward only.
    - Reverse traversal or random access is not allowed.
    - But cursor can be closed and then re-opened (from start).
  - Asensitive

- When cursor is opened, the address of rows (as per select clause) are recorded into cursor and then accessed one by one.
- If any client changes any of the row, while cursor is still traversing; the changes by that client will be immediately available into (accessible by) cursor.
- Internally it is not processing copy of data/rows.
- Hence MySQL cursors are faster.

```
-- Q. insert all empno & dname into result table whose sal is greater than avg sal in asc order of
dname.

        DECLARE v_cur CURSOR SELECT e.empno, d.dname FROM EMP e INNER JOIN DEPT d ON d.deptno =
e.deptno WHERE e.sal > (SELECT AVG(sal) FROM EMP) ORDER BY d.dname;

-- OR

        DELCLARE v_avgsal DOUBLE;
        SELECT AVG(sal) INTO v_avgsal FROM EMP;
        DECLARE v_cur CURSOR SELECT e.empno, d.dname FROM EMP e INNER JOIN DEPT d ON d.deptno =
e.deptno WHERE e.sal > v_avgsal ORDER BY d.dname;
```

Functions

- User defined functions are routines stored on server and are used in queries (DQL or DML).
- UDFs are written to provide functionalities which are not available in built-in SQL functions.
- In MySQL, UDFs are single row functions. (Implementing group UDF is out of scope).
- Note that UDFs are slower than built-in functions.
- UDF are similar to SP, but UDF return value.
- In MySQL, UDF cannot have OUT or INOUT params. Here all params are IN params by default.
- Functions are not called using CALL() statement. They are called in SQL queries. They will be executed for each row.
- There are two types of functions in MySQL:
  - Deterministic
    - The return value is only dependent on its params.
    - Not dependent on current time or state of table.
    - MySQL stores/cache params & result, so that if fn is called again with same params result returned quickly (without actually calling fn).
  - Non-deterministic
    - The return value is dependent on its params as well as some more factors.
    - These factors are current time or state of table.

- MySQL doesn't cache its results. Function is always called.

```
CREATE FUNCTION fn_name(p1 DATATYPE, p2 DATATYPE, ...)
RETURNS DATATYPE
[NOT] DETERMINISTIC
BEGIN
        fn body; -- PL syntax
        RETURN ret_value;
END;
```

- All stored proc & functions are stored into system table information_schema.routines;

```
DESC information_schema.routines;

SELECT * FROM information_schema.routines;

SELECT * FROM information_schema.routines \G

SELECT ROUTINE_SCHEMA, ROUTINE_TYPE, ROUTINE_NAME, DEFINER, ROUTINE_DEFINITION FROM
information_schema.routines \G
```

Triggers

- Triggers are also MySQL programs. Hence written in same PL syntax and stored on server. They execute faster than SQL queries.
- Triggers are never called explicitly. Their execution is triggered on some events.
- Triggers can be invoked before (pre) or after (post) DML operations.
  - BEFORE INSERT
  - AFTER INSERT
  - BEFORE UPDATE
  - AFTER UPDATE
  - BEFORE DELETE
  - AFTER DELETE
- You should not write TCL statements in triggers. Triggers are always executed in the tx, in which DML operation is in progress.
- If trigger fails, the current tx is rollbacked.
- Triggers don't have parameters. In MySQL, triggers have two special keywords "NEW" and "OLD", which represent new row and old row respectively.
- INSERT trigger have only NEW keyword (OLD is not applicable).
- DELETE trigger have only OLD keyword (NEW is not applicable).

- UPDATE trigger have NEW & OLD keyword.
- Trigger is executed for each row (inserted, updated or deleted).

```
CREATE TRIGGER trig_name
trigger_type ON table_name
FOR EACH ROW
BEGIN
        trigger body;
        use OLD & NEW keywords here as appropriate;
END;
```

- Triggers can be cascaded.
- Circular cascading of trigger will stop execution of triggers. It is called as "Mutating table error".
- Triggers are stored in information_schema.triggers table.

```
SELECT * FROM information_schema.triggers;

SHOW TRIGGERS FROM classwork;
```

- Applications:
  - DML operations logging (Audit trails).
  - To store old data (History table).
  - To create backup table (Shadow table).
  - To replicate data on other server.
  - Data cleansing before inserting/updating data.

Temporary Tables

- View is limited view of data, as per given SELECT statement. But internally no memory/space allocated to view. Any operation done on view will execute on main table. Advantages of view are security, hiding source code and simplifying queries. View doesn't affect performance of query.
- Feature of MySQL.
- Can create new **temporary** table per user session to store/cache the result of SELECT query.
- They are not visible & accessible to other user. Even you cannot use GRANT/REVOKE for doing so.
- This table is auto deleted when user session is completed (MySQL terminal is closed).
- Using this we can simplify some complex queries and at the same time we can make it more efficient.

- Any changes done in main table after creation of temp table are not available into temp table.
- Any changes done in temp table are not available main temp table.
- Temp table can have same name as of existing db table. In this case you access temp table. To access main table, first delete temp table. It is not good practice to give same name to avoid confusion.

```
CREATE TEMPORARY TABLE t_deptemp
AS SELECT e.empno, e.ename, e.sal, d.deptno, d.dname
FROM EMP e INNER JOIN DEPT d ON e.deptno = d.deptno;

SHOW TABLES;

SELECT * FROM t_deptemp;

DROP TEMPORARY TABLE t_deptemp;
```

Codd's Rules

- Rule-0: For any system that is advertised as, or claimed to be, a relational data base management system, that system must be able to manage data bases entirely through its relational capabilities.

- Rule-1: All information in a relational data base is represented explicitly at the logical level and in exactly one way – by values in tables. (Tables + Rows + Columns.)

- Rule-2: Each and every datum (atomic value) in a relational data base is guaranteed to be logically accessible by resorting to a combination of table name, primary key value and column name. (Primary Key for each Row).

- Rule-3: Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type.

- Rule-4: The data base description (metadata) is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data. (Stored into system tables e.g. mysql.user, information_schema.routines, information_schema.triggers).

- Rule-5: A relational system may support several languages and various modes of terminal use (for example, the fill-in-the-blanks mode). However, there must be at least one language

whose statements are expressible, per some well-defined syntax, as character strings. (SQL language).

- Rule-6: All views that are theoretically updatable are also updatable by the system.
- Rule-7: The capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update and deletion of data.
- Rule-8: Application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representations or access methods. (Storage engine)
- Rule-9: Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit unimpairment are made to the base tables. (Views)
- Rule-10: Integrity constraints specific to a particular relational data base must be definable in the relational data sublanguage and storable in the catalog, not in the application programs. (Foreign-key constraint)
- Rule-11: The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. (Clustering)
- Rule-12: If a relational system has a low-level (single-record-at-a-time) language, that low level cannot be used to subvert or bypass the integrity rules and constraints expressed in the higher level relational language (multiple-records-at-a-time). (Snapshot)

Normalization

- Concept of table design --> Table, Structure, Data Types, Width, Constraints, Relations.
- Goals:
    - Efficient table structure
    - Avoid data redundancy i.e. unnecessary duplication of data (to save disk space)
    - Reduce problems of insert, update & delete.
- Done from input perspective.

- Based on user requirements.
- Part of software design phase.
- View entire application on per transaction basis & then normalize each transaction separately.
- Transaction Examples:
  - Banking: Open New Account, Deposit Amount, Withdraw Amount.
  - Rail Reservation: Reservation, Cancellation.
  - Online Shopping: Customer Order, Stock Update, New Product, ...

Getting ready for Normalization:

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

UNF --> starting point of normalization.

- Delete anomaly
- Update anomaly
- Insert anomaly

Normalization steps:

1. Remove repeating group into a new table.
2. Key elements will be PK of new table.

3. (Optional) Add PK of original table to new table to give us Composite PK.
    - Repeat steps 1-3 infinitely -- to remove all repeating groups into new tables.
    - This is 1-NF. No repeating groups present here. One to Many relationships between two tables.
4. Only table with composite PK to be examined.
5. Those columns that are not dependent on the entire composite PK, they are to be removed into a new table.
6. The key elements on which the non-key elements were originally dependent, it is to be added to the new table, and it will be the PK of new table.
    - Repeat steps 4-6 infinitely -- to separate all non-key elements from all tables with composite primary key.
    - This is 2-NF. Many-to-Many relationship.
7. Only non-key elements are examined for inter-dependencies.
8. Inter-dependent columns that are not directly related to PK, they are to be removed into a new table.
9. (a) Key element will be PK of new table.

    (b) The PK of new table is to be retained in original table for relationship purposes.
    - Repeat steps 7-9 infinitely to examine all non-key elements from all tables and separate them into new table if not dependent on PK.
    - This is 3-NF.

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

MySQL client software's

1. mysql SQL shell

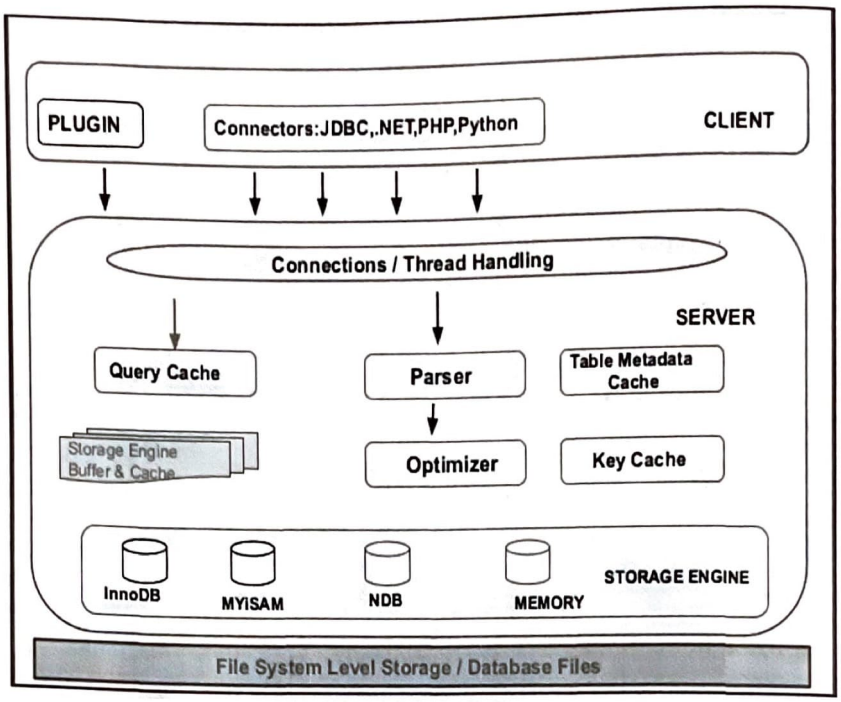
2. mysql workbench (GUI desktop based tool)
    - sudo apt-get install mysql-workbench
3. phpmyadmin (GUI web based tool)
    - sudo apt-get install phpmyadmin

MySQL Architecture

Physical architecture

- Configuration files
    - /etc/mysql/my.cnf
- Installed Files
    - Executable files
        - mysqld
        - mysqladmin
        - mysql
        - mysqldump
        - ...
    - Log files
        - pid files
        - socket files
        - document files
        - libraries
- Data Files
    - Data directory
        - Server logs
        - Status files
        - Innodb tablespaces & log buffer
    - Database directory files
        - Data & Index files (.ibd)
        - Object structure files (.frm, .opt)

Logical architecture



- Client
- Server (mysqld)
  - Accept & process client requests
  - Multi-threaded process
  - Dynamic memory allocation
    - Global allocation
    - Session allocation
- Parser
  - SQL syntax checking.
  - Generate sql_id for query.
  - Check user authentication.
- Optimizer
  - Generate efficient query execution plan
  - Make use of appropriate indexes
  - Check user authorization.
- Query cache

  - Server level (global) cache
  - Speed up execution if identical query is executed previously
- Key cache
  - Cache indexes
  - Only for MyISAM engine
- Storage engine
  - Responsible for storing data into files.
  - Features like transaction, storage size, speed depends on engine.
  - Supports multiple storage engines
  - Can be changed on the fly (table creation)
  - Important engines are InnoDB, MyISAM, NDB, Memory, Archive, CSV.

InnoDB engine

- Fully transactional ACID.
- Row-level locking.
- Offers REDO and UNDO for transactions.
- Shared file to store objects (Data and Index in the same file - .ibd)
- InnoDB Read physical data and build logical structure (Blocks and Rows)
- Logical storage called as TABLESPACE.
- Data storage in tablespace
  - Multiple data files
  - Logical object structure using InnoDB data and log buffer

MyISAM

- Non-transactional storage engine
- Table-level locking
- Speed for read
- Data storage in files and use key, metadata and query cache
  - .frm for table structure
  - .myi for table index
  - .myd for table data

NDB

- Fully Transactional and ACID Storage engine.
- Row-level locking.
- Offers REDO and UNDO for transactions.
- NDB use logical data with own buffer for each NDB engine.
- Clustering: Distribution execution of data and using multiple mysqld.

Clustering in MySQL

- Using multiple machines to store & process data.
- All machines are connected through high-speed network.
- Two types of clusters are supported.
  - InnoDB cluster
  - NDB cluster

InnoDB cluster

- Designed for High-Availability
- System continue to function even if one of the server goes down
- Consists of multiple servers running **mysqld** and a **router**.
- Group replication
  - Multiple servers maintains same copy of the data.
  - Usually one of the server is primary (i.e. handles write+read)
  - Remaining all servers are secondary (i.e. only read)
  - If primary fails, one of the secondary is elected as primary.
- Router
  - Usually installed on same machine as of client application.
  - Client fire SQL queries to router.
  - Router send them to appropriate server from replication group.
  - Act as load balancer.
  - Use separate ports for write & read (in config files).
- MySQL shell
  - Used for creating & monitoring cluster

- It is different from SQL shell -- mysqlsh
- dba.createCluster()
- cluster = dba.getCluster()
- cluster.status()

NDB cluster

- Designed for High-Availability (99.99% up time)
- Management node (ndb_mgmd)
  - manage the other nodes within the NDB Cluster
  - providing configuration data, starting and stopping nodes, and running backups.
  - node of this type should be started first
- Data node (ndbd)
  - stores cluster data
  - multiple nodes for replica per fragment
  - minimum two replicas per fragment for HA
  - cluster tables are stored completely in memory rather than on disk
- SQL/API node (mysqld)
  - accesses the cluster data
  - a specialized type of API node
  - minimum two replicas are recommended

MySQL Snapshots

- Snapshot is state of database. It includes table metadata & data.
- Applications:
  - Porting existing appln on new server.
  - Recovering from database crash/corruption.
- Backup & Restore.
- Methods of snapshots:
  - MySQL workbench -- export & import
  - PhpMyAdmin -- export & import
  - MySQL utilities -- mysqldump & mysql
  - Raw files backup & restore -- very fast, difficult

mysqldump & mysql

```
mysqldump -u [username] -p[password] [database_name] > [dump_file.sql]

mysqldump -u [username] -p[password]  [dbname1,dbname2...] > [dump_file.sql]

mysqldump -u [username] -p[password] --all-database > [dump_file.sql]

mysql -u [uname] -p[pass] [db_to_restore] < [backupfile.sql]
```

Mongo DBMS

- Performs CRUD operations on data.
- In RDBMS, data is stored in **tables** in **rows**. Schema is defined by its **columns**.
- In Mongo, data is stored in **collections** in **documents**. Schema is flexible. Each document can have multiple/different **fields**.
- Mongo documents are in **JSON** (Java Script Object Notation) format.
- JSON is a text format to represent the data. It is used heavily nowdays at many places (where earlier people were using XML).
- JSON example:

```
{
        "_id": 1,
        "name": "Nilesh Ghule",
        "age": 35,
        "address": {
                "area": "Katraj",
                "city": "Pune",
                "pin": 411046
        },
        "contact": {
                "email": "nilesh@sunbeaminfo.com",
                "mobile": "9527331338"
        },
        "rating": 3.8,
        "isTrainer": true,
        "skills": [ "Database", "Hadoop", "Spark", "Device Drivers", "Micro-controllers" ],
        "birth": ISODate("1983-09-28")
}
```

- JSON is internally converted into BSON (Binary JSON) before storing into mongo db. For efficient storage & processing. In BSON, each data type is represeted as a number e.g. array is 4, null is 10, ...
- In Mongo, max size of one document is 16 MB.
- Each mongo doc should have an unique "_id". User can provide unique id for each record (like PK in RDBMS).
- If user have not given "_id", mongo will auto generate it of type ObjectId of 12 bytes.
  - counter -- arbitrary counter.
  - timestamp -- current client time.
  - process id -- client (OS) process id.
  - machine -- client machine id.

```
command> mongo
show databases;

use kdac;

show collections;
db.persons.insert({
        "email": "nilesh@sunbeaminfo.com",
        "mobile": "9527331338",
        "name": "Nilesh",
        "age": 35
});

db.persons.insert({
        "name": "Nitin",
        "age": 40,
        "mobile": "9881208115"
});

db.persons.insert({
        "name": "Prashant",
        "mobile": "9881208114",
        "email": "prashant@sunbeaminfo.com"
});

db.persons.insert({
        "name": "Sandeep",
        "addr": "Katraj, Pune"
});

db.persons.find();
// Cursor is returned by find() method of collection.
// Cursor methods: pretty(), sort(), limit(), skip(), ...

db.persons.find().pretty();

db.persons.find().limit(2);

db.persons.find().skip(2).limit(1);

db.persons.find().sort({"name": -1});
// asc order of name
// copy all lines from empdept_separate.js on mongo shell

show collections;

db.emp.find();

db.emp.find().sort({"deptno": 1});

db.emp.find().sort({"deptno": 1, "sal": -1});
// multi-level sorting
```

```
//db.colname.find(criteria, projection);
db.emp.find({}, {"ename":1, "sal":1});

db.emp.find({}, {"job":0, "mgr":0, "hire": 0});

db.emp.find({}, {"ename":1, "sal":1, "deptno":0}); //error
//db.colname.find(criteria, projection);
db.emp.find({"_id": 7900});

db.emp.find({"deptno": 10});

//Q. find emp with max sal in dept 20.
db.emp.find({"deptno": 20});
sort({"sal": -1}).
limit(1).
pretty();

// relational operators: $eq, $ne, $gt, $gte, $lt, $lte, $in, $nin, ...
// db.colname.find({ "fieldname" : { "$oper" : value } });
db.emp.find({"sal": { $lte : 1000 } });
// logical operators: $and, $or, $nor
// db.colname.find({ "$log_oper" : [
//                 { "fieldname" : { "$reln_oper" : value } },
//                 { "fieldname" : { "$reln_oper" : value } }
// ] });

// Q. find all salesman with sal >= 1400.
db.emp.find({
        $and : [
                { "job" : "SALESMAN" },
                { "sal" : { $gte : 1400 } }
        ]
});

// Q. find all salesman or sal >= 3000.
db.emp.find({
        $or : [
                { "job" : "SALESMAN" },
                { "sal" : { $gte : 3000 } }
        ]
});

// Q. find all salesman & clerks.
db.emp.find({
        $or : [
                { "job" : "SALESMAN" },
                { "job" : "CLERK" }
        ]
});

// Q. find all salesman & clerks using $in.
db.emp.find({
        "job": { $in : [ "SALESMAN", "CLERK" ] }
```

```
});
db.emp.insert({"_id": 1, "ename": "Soham", "sal": 3400});
db.emp.insert({"_id": 2, "ename": "Rohan", "sal": 2400});
db.emp.insert({"_id": 3, "ename": "Ashwin", "sal": 2600});

db.emp.find();

//db.colname.remove(criteria);
db.emp.remove({"_id": 3});

db.emp.remove({}); // delete all emp

db.persons.drop(); // delete all records + collection

show collections;
//db.colname.update(criteria, new_record);
db.emp.update({"_id": 2}, { "sal": 2500 });

//db.colname.update(criteria, changes);
db.emp.update({"_id": 1}, {
        $set: { "sal" : 3500 }
});
//db.colname.update(criteria, changes, is_upsert);
db.emp.update({"_id": 4, "ename": "Vishal"}, {
        $set: { "sal": 3200, "deptno": 50 }
}, false);
// no record found, hence no record modified.

db.emp.update({"_id": 4, "ename": "Vishal"}, {
        $set: { "sal": 3200, "deptno": 50 }
}, true);
// if record is found, it will be updated.
// if record not found, record of given criteria is inserted & then its given fields will be
edited. This is called as "upsert".

db.emp.find();
// aggregation pipeline
// db.colname.aggregate([
//              { stage1 },
//              { stage2 },
//              { stage3 },
//              ...
// ]);
// stages/operators: $group, $match, $sort, $project, $lookup, $out, ...
db.emp.aggregate([
{
        $group : {
                "_id": "$job",
                "total": { $sum : "$sal" }
        }
}
]);
```

Mongo Indexes

- To speedup execution of find() i.e. searching is faster.
- By default all collections are indexed on "_id" field.
- Types of indexes:
  - Index
  - Unique index
  - Composite index
  - Geo-spatial index
  - TTL index

```
db.emp.createIndex({"deptno": 1});

db.emp.createIndex({"sal": -1});

db.emp.createIndex({"deptno": 1, "job": 1});

db.dept.createIndex({"dname": 1}, {"unique": true});
// collections are auto created, while insert record.
// they can be created explicitly using createCollection().
db.createCollection("stud");

db.stud.insert(...);
```

Mongo Data Modeling

- How data is organized into mongo database.
- Two modeling methods:
  - Embedded data model
  - Referenced data model (Normalized data model)

Embedded data model

- The related objects are embedded into some parent object.
- example:
  - emp collection:
    - _id, ename, sal, mgr, comm, job, hire, dept
    - dept is a nested object that contains _id, dname, loc

Reference data model

- Data is in separate collections.
- The documents are connected by some field.
- example:
  - dept collection
    - _id, dname, loc
  - emp collection
    - _id, ename, sal, mgr, comm, job, hire, deptno
    - Here "deptno" field is mapped to "_id" of dept.
- Reduces data redundancys

Mongo Consistency

- Each record editing is always atomic.
  - Multi-row tx are not straight forward.
  - Specialized design patterns can be used for such tx e.g. two phase commit.
- Changes done by one client are available to all clients.
- Number of clients can work simultaneously.
- All changes are saved on disk.