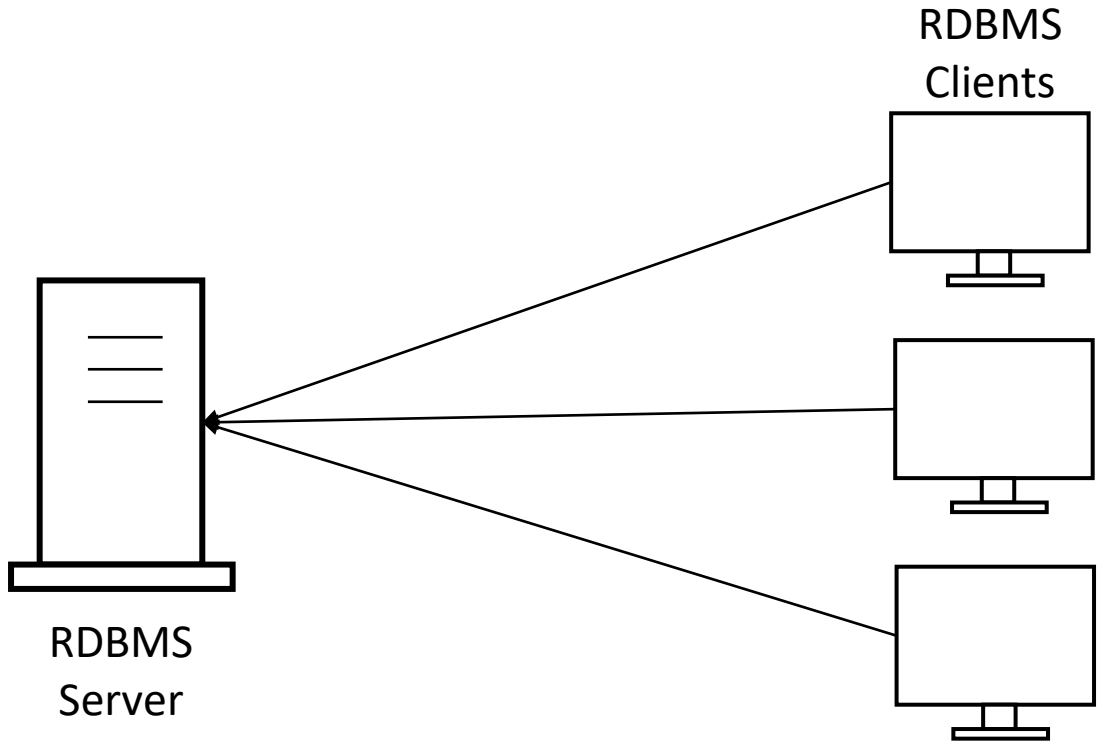
# DBMS

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

# RDBMS

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



# SQL

- Clients send SQL queries to RDBMS server and operations are performed accordingly.
- Originally it was named as RQBE (Relational Query By Example).
- SQL is ANSI standardised in 1987 and then revised multiple times adding new features.
- SQL is case insensitive.
- There are five major categories:
  - DDL: Data Definition Language e.g. CREATE, ALTER, DROP, RENAME.
  - DML: Data Manipulation Language e.g. INSERT, UPDATE, DELETE.
  - DQL: Data Query Language e.g. SELECT.
  - DCL: Data Control Language e.g. CREATE USER, GRANT, REVOKE.
  - TCL: Transaction Control Language e.g. SAVEPOINT, COMMIT, ROLLBACK.
- Table & column names allows alphabets, digits & few special symbols.
- If name contains special symbols then it should be back-quotes.
- e.g. Tbl1, `T1#`, `T2$` etc. Names can be max 30 chars long.

# MySQL

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

# Getting started

- root login can be used to perform CRUD as well as admin operations.
- terminal> mysql –u root –pmanager mydb
  - mysql> SHOW DATABASES;
  - mysql> SELECT DATABASE();
  - mysql> USE mydb;
  - mysql> SHOW TABLES;
  - mysql> CREATE TABLE student(id INT, name VARCHAR(20), marks DOUBLE);
  - mysql> INSERT INTO student VALUES(1, 'Abc', 89.5);
  - mysql> SELECT * FROM student;

# Database logical layout

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

# Agenda

```
DBMS VS RDBMS
SQL
SQL Categories
MySQL
Getting Started with MySQL
Database- Logical & Physical Layout
SQL Scripts
```

# DBMS vs RDBMS

```
- DBMS -> File based System
- RDBMS -> Client Server based
    - Relational DataBase Management System
```

# SQL

```
- Structured Query Language
- RQBE -> Relational Query By Example
- ANSI standarized in 1987 to SQL
- SQL is case insensitive
```

# SQL categories

```
1. DDL -> Data Defination Language
    - CREATE,ALTER,DROP,RENAME,TRUNCATE
2. DML -> Data Manipulation Language
    - INSERT,UPDATE,DELETE
3. DQL -> Data Query Language
    - SELECT
4. DCL -> Data Control Language
    - CREATE USER,GRANT,REVOKE
5. TCL -> Transaction Control Language
    - SAVEPOINT,COMMIT,ROLLBACK
```

# MySQL

```
MySQL was developed by Micheal Wideneus in 1995.
Named after his daughter 'Myia'
```

```
Sun Microsystems took over Mysql in 2008.
Oracle took over sunmicrosystems in 2010.
Mysql is free and open source under GPL.
```

## Getting Started.

```
mysql -u root -p
    -u -> username
    -p -> password
-If you get an error saying can't connect to Mysql check if server is RUNNING OR
STOPPED
- mysql is not recogniozed command then set the path for mysql in environment
variables.
- Mysql gets installed in your machine at below locations
C:\Program Files\MySQL\MySQL Server 8.0
C:\Program Data\MySQL\MySQL Server 8.0
- C:\Program Files\MySQL\MySQL Server 8.0\bin ->Copy this path and paste it in
your environment variables.
```

## Database

```
    mysql -u root -p

    SELECT DATABASE(); -- Null

    SHOW DATABASES; -- list of all exixting databases

    CREATE DATABASE sunbeam_students; --to create a new database

    SHOW DATABASES; -- you found your created db

    SELECT DATABASE(); -- NULL

    USE sunbeam_students; --To select the database for working

    SELECT DATABASE(); -- sunbeam_students
```

## TABLES

```
    CREATE TABLE students(
        regno INT,
        name CHAR(20),
        marks DOUBLE
    );
```

```
CREATE TABLE students_Group(
        regno INT,
        groupname CHAR(10)
    );

    DESCRIBE students; -- see the table structure

    SHOW TABLES; -- dispaly all the tables available in your selected database

    DROP DATABASE sunbeam_stduents;
```

## SQL SCRIPTS

```
-- import the classwork database into your mysql

CREATE DATABASE classwork;
USE classwork;
SELECT DATABASE();
SHOW TABLES;

SOURCE <drag drop path to the .sql file> ;
SHOW TABLES;

SELECT * FROM emp;
SELECT * FROM dept;
```

## Add and view Data From table

```
CREATE DATABASE my_db;
USE mydb;

CREATE TABLE students(
        regno INT,
        name CHAR(20),
        marks DOUBLE
    );

SHOW TABLES;

DESC students;

--to add the data into table
INSERT INTO students VALUES(1,'stu1',78);

--to display all data from table
SELECT * FROM studnets;
```

# CHAR vs VARCHAR vs TEXT

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

# INSERT – DML

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

# SQL scripts

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

# SELECT – DQL

- Select all columns (in fixed order).
  - SELECT * FROM table;
- Select specific columns / in arbitrary order.
  - SELECT c1, c2, c3 FROM table;
- Column alias
  - SELECT c1 AS col1, c2 col2 FROM table;
- Computed columns.
  - SELECT c1, c2, c3, expr1, expr2 FROM table;

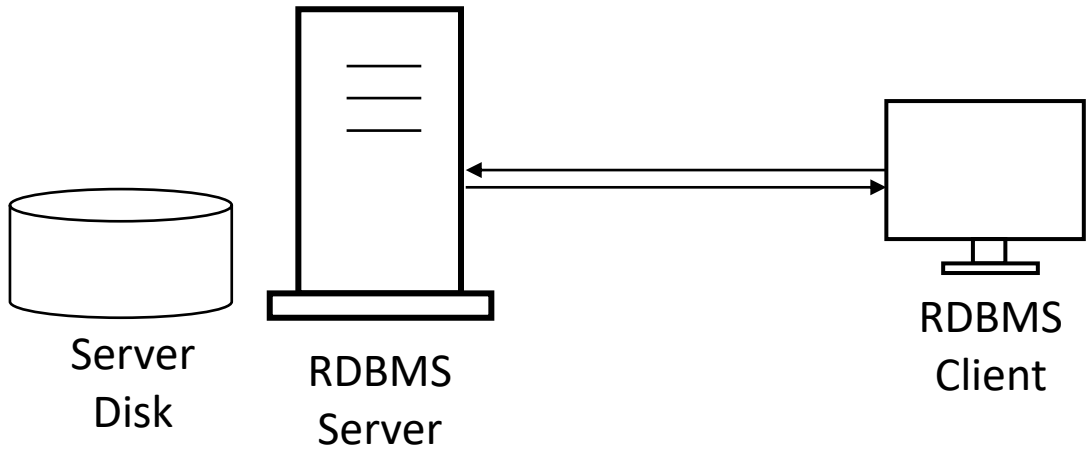
```
SELECT c1,
CASE WHEN condition1 THEN value1,
WHEN condition2 THEN value2,
...
ELSE valuen
END
FROM table;
```

# SELECT – DQL

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

# SELECT – DQL – ORDER BY

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



# Agenda

```
- Data Types
- Char vs Varchar vs TEXT
- DQL
- Computed Columns
- Distinct
- LIMIT
- Order By
- WHERE clause
- Relational Operators
- IN,BETWEEN Operator
```

# Steps to change mysql cmd Prompt

```
- Open environment variable
- Inside system variable for rohan click on new
- Enter Variable name as MYSQL_PS1
- Enter value as W1_ROHAN_12345> (Group_name_rollno)
- Click on OK.
```

# Datatypes

```
1. Numeric Type
    - tinyint(1 byte)
    - smallint(2 bytes)
    - mediumint(3 bytes)
    - int(4 bytes)
    - bigint(8 bytes)
    - float(4 bytes)
    - double(8 bytes)
    - Decimal(m,n)
        m-> total no of digits
        n-> no of digits after the decimal point
        e.g DECIMAL(4,2) = 12.30
        e.g DECIMAL(5,4) = 1.2345
2. String Type
    - CHAR(n) -> n is the no of characters you want
    - VARCHAR() ->n is the no of characters you want
    - TINYTEXT(255)
    - TEXT(65K)
    - MEDIUMTEXT(16MB)
    - LONGTEXT(4GB)
3. Binary Type
```

```
    - TINYBLOB,BLOB,MEDIUMBLOB,LONGBLOB
4. Misc Type
    - ENUM (M,F,O)-> Radiobutton
    - SET (C,CPP,JAVA) ->Checkbox
5. DateTime Type
    - DATE -> yyyy-mm-dd (1000-01-01 to 9999-12-31)
    - TIME -> hr:min:sec (838:59:59)
    - DATETIME -> yyyy-mm-dd hr:min:sec (1000-01-01 to 9999-12-31)
                                        (00:00:00 to 23:59:59)
    - YEAR - 1901 to 2155
```

## Char vs Varchar vs TEXT

```sql
CREATE TABLE temp(
   c1 CHAR(4),
   c2 VARCHAR(4),
   c3 TEXT(4)
);

 DESC temp;

INSERT INTO temp VALUES('ab','ab','ab');
SELECT * FROM temp;

INSERT INTO temp VALUES('abc','abc','abc');
SELECT * FROM temp;

INSERT INTO temp VALUES('abcd','abcd','abcd');
SELECT * FROM temp;

INSERT INTO temp VALUES('abcde','abcd','abcd');--error

INSERT INTO temp VALUES('abcd','abcde','abcd');--error

INSERT INTO temp VALUES('abcd','abcd','abcde');

INSERT INTO temp VALUES('abcde','abcde','abcde');
```

## DQL -> SELECT

```sql
USE classwork;
SELECT DATABASE();
SHOW TABLES;

SELECT * FROM emp;
SELECT * FROM dept;
```

```
--dispaly emp name,job,sal from emp table
SELECT ename,job,sal FROM emp;
SELECT ename,sal,job FROM emp;
```

```
--add new emp with name as Rohan sal as 3000 and job as analyst
INSERT INTO emp VALUES('Rohan',3000,'ANALYST');--error
INSERT INTO emp(ename,sal,job) VALUES('Rohan',3000,'ANALYST');

--add 2 new emps in single query
INSERT INTO emp(ename,sal,job)VALUES
('Pratik',2000,'ANALYST'),
('Onkar',2500,'CLERK');
```

## DQL- Computed Column

```
-- give DA as an allowance for all employees. It should be 50% of sal
-- dispaly all emps along witn their allowance

--Computed Column
SELECT ename,sal,sal*.50 FROM emp;

--Column Alias
SELECT ename,sal,sal*.50 AS DA FROM emp;

--display gross_sal = sal+DA from emp.
SELECT ename,sal,sal*0.5 AS DA, sal+sal*0.5 AS gross_sal FROM emp;

SELECT ename,sal,sal*0.5 AS DA, sal+DA AS gross_sal FROM emp; --error

--dispaly emp and their deptnames
SELECT ename,deptno, CASE
WHEN deptno=10 THEN 'ACCOUNTING'
WHEN deptno=20 THEN 'RESEARCH'
WHEN deptno=30 THEN 'SALES'
END
AS dept_name
FROM emp;
```

## DQL -> Distinct

```
--display all unique jobs from emp.
SELECT job FROM emp;
--It will fetch all jobs from emp table along with repetation

SELECT DISTINCT job FROM emp;
```

```
--display all unique deptno from emp.
SELECT deptno FROM emp;

SELECT DISTINCT deptno FROM emp;

--display unique jobs from each dept
SELECT DISTINCT deptno,job FROM emp;
```

## DQL-> Limit

```
--display only 5 employees
SELECT * FROM emp;
SELECT * FROM emp LIMIT 5;

--display only 10 employees
SELECT * FROM emp LIMIT 10;

--display only 5 rows with emp name,sal and job from emp
SELECT ename,sal,job FROM emp;
SELECT ename,sal,job FROM emp LIMIT 5;

--diaply empname,sal,comm from emp skip 5 rows and show 3 rows after that
SELECT ename,sal,comm FROM emp;
SELECT ename,sal,comm FROM emp LIMIT 5;
SELECT ename,sal,comm FROM emp LIMIT 5,3;
```

## Order BY

```
-- dispaly emp with sal sorted in ascending manner
SELECT * FROM emp;
SELECT * FROM emp ORDER BY sal; --Default sorting is Ascending

-- dispaly emp with deptno sorted in ascending manner
SELECT * FROM emp;
SELECT * FROM emp ORDER BY deptno;

-- dispaly emp with sal sorted in descending manner
SELECT * FROM emp ORDER BY sal DESC;

-- dispaly emps sorted on their deptno and their jobs
SELECT ename,deptno,job FROM emp;
SELECT ename,deptno,job FROM emp ORDER BY deptno,job;

-- dispaly emps sorted on their deptno asc and their jobs in desc
SELECT ename,deptno,job FROM emp ORDER BY deptno ,job DESC;
```

## ORDER By using Limit

```
--display top 3 emps as per highest sal
SELECT * FROM emp;
SELECT * FROM emp ORDER BY sal DESC;
SELECT * FROM emp ORDER BY sal DESC LIMIT 3;

--display single emp which comes last in alphabetical order.
SELECT * FROM emp;
SELECT * FROM emp ORDER BY ename;
SELECT * FROM emp ORDER BY ename DESC;
SELECT * FROM emp ORDER BY ename DESC LIMIT 1;

--display single emp with lowest salary
SELECT * FROM emp ORDER BY sal LIMIT 1;

--display single emp with 3rd lowest salary
SELECT * FROM emp ORDER BY sal;
SELECT * FROM emp ORDER BY sal LIMIT 2,1;

--dispaly single emp with 2nd highest salaray
SELECT * FROM emp ORDER BY sal DESC;
SELECT * FROM emp ORDER BY sal DESC LIMIT 1,1;

--dispaly empname,DA from emop sorted based on DA
SELECT ename,sal*0.5 AS DA FROM emp;
SELECT ename,sal*0.5 FROM emp ORDER BY sal*0.5;
SELECT ename,sal*0.5 AS DA FROM emp ORDER BY DA;
SELECT ename,sal*0.5 FROM emp ORDER BY 2;
```

## WHERE CLAUSE

```
-- dispaly emps from dept no 30
SELECT * FROM emp;
SELECT * FROM emp WHERE deptno=30;

--display all emps with sal<2000
SELECT * FROM emp;
SELECT * FROM emp WHERE sal<2000;

--display all emp working as ANALYST
SELECT * FROM emp;
SELECT * FROM emp WHERE job="ANALYST";

--display all emps not working in dept 30
SELECT * FROM emp;
SELECT * FROM emp WHERE deptno!=30;
```

```sql
SELECT * FROM emp WHERE deptno<>30;
SELECT * FROM emp WHERE NOT deptno=30;

--dispaly all emp who are not salesman
SELECT * FROM emp WHERE job = 'SALESMAN';
SELECT * FROM emp WHERE job != 'SALESMAN';
SELECT * FROM emp WHERE job <> 'SALESMAN';
SELECT * FROM emp WHERE NOT job = 'SALESMAN';

-- display all emp who work as manager and analyst
SELECT * FROM emp;
SELECT * FROM emp WHERE job = 'MANAGER';
SELECT * FROM emp WHERE job = 'ANALYST';
SELECT * FROM emp WHERE job = 'MANAGER' OR job = 'ANALYST';
SELECT * FROM emp WHERE job IN ('MANAGER','ANALYST');

--display all emp in sal range 1000 to 2000
SELECT * FROM emp;
SELECT * FROM emp WHERE sal>=1000;
SELECT * FROM emp WHERE sal<=2000;
SELECT * FROM emp WHERE sal>=1000 AND sal<=2000;
SELECT * FROM emp WHERE sal BETWEEN 1000 AND 2000;

--display emps hired in 1981
SELECT * FROM emp;
SELECT * FROM emp WHERE hire='1981';--error
--1981-01-01 -> 1981-12-31
SELECT * FROM emp WHERE hire>='1981-01-01';
SELECT * FROM emp WHERE hire<='1981-12-31';
SELECT * FROM emp WHERE hire>='1981-01-01' AND hire<='1981-12-31';
SELECT * FROM emp WHERE hire BETWEEN '1981-01-01' AND '1981-12-31';
```

# SELECT – DQL – WHERE

- It is always good idea to fetch only required rows (to reduce network traffic).
- The WHERE clause is used to specify the condition, which records to be fetched.
- Relational operators
  - <, >, <=, >=, =, != or <>
- NULL related operators
  - NULL is special value and cannot be compared using relational operators.
  - IS NULL or <=>, IS NOT NULL.
- Logical operators
  - AND, OR, NOT

# SELECT – DQL – WHERE

- BETWEEN operator (include both ends)
  - c1 BETWEEN val1 AND val2
- IN operator (equality check with multiple values)
  - c1 IN (val1, val2, val3)
- LIKE operator (similar strings)
  - c1 LIKE ‘pattern’.
  - % represent any number of any characters.
  - _ represent any single character.

# UPDATE – DML

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

# DELETE – DML vs TRUNCATE – DDL vs DROP – DDL

- DELETE
  - To delete one or more rows in a table.
  - Delete row(s)
    - DELETE FROM table WHERE c1=value;
  - Delete all rows
    - DELETE FROM table
- TRUNCATE
  - Delete all rows.
    - TRUNCATE TABLE table;
  - Truncate is faster than DELETE.
- DROP
  - Delete all rows as well as table structure.
    - DROP TABLE table;
    - DROP TABLE table IF EXISTS;
  - Delete database/schema.
    - DROP DATABASE db;

# Seeking HELP

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

# DUAL table

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

# SQL functions

- RDBMS provides many built-in functions to process the data.
- These functions can be classified as:
  - Single row functions
    - One row input produce one row output.
    - e.g. ABS(), CONCAT(), IFNULL(), …
  - Multi-row or Group functions
    - Values from multiple rows are aggregated to single value.
    - e.g. SUM(), MIN(), MAX(), …
- These functions can also be categorized based on data types or usage.
  - Numeric functions
  - String functions
  - Date and Time functions
  - Control flow functions
  - Information functions
  - Miscellaneous functions

# Numeric & String functions

- ABS()
- POWER()
- ROUND(), FLOOR(), CEIL()

- ASCII(), CHAR()
- CONCAT()
- SUBSTRING()
- LOWER(), UPPER()
- TRIM(), LTRIM(), RTRIM()
- LPAD(), RPAD()
- REGEXP_LIKE()

# Date-Time and Information functions

- VERSION()
- USER(), DATABASE()

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

## Agenda

```
- Where clause
    - LIMIT
    - LIKE
- DML - Update,Delete
- DDL - Drop,Truncate
- DUAL
- SQL FUNCTIONS
    - String
    - Numeric
    - Date and Time Functions
```

## Where Clause for NULL Condition

```
--display all emps with comm as null
SELECT * FROM emp;
SELECT * FROM emp WHERE comm=NULL; -- empty set
SELECT * FROM emp WHERE comm IS NULL;
SELECT * FROM emp WHERE comm <=> NULL;

--display all emps with comm as not null
SELECT * FROM emp WHERE comm!=NULL; -- empty set
SELECT * FROM emp WHERE comm IS NOT NULL;
SELECT * FROM emp WHERE NOT (comm IS NULL);
```

## Extra Examples

```
--Insert into emp 3 new emps with name as 'B','J' and 'K'
INSERT INTO emp(ename) VALUES('B'),('J'),('K');

--dispaly all emps whose first letter of name is in the range of B to J
SELECT * FROM emp;
SELECT * FROM emp WHERE ename>='B';
SELECT * FROM emp WHERE ename<='J';
SELECT * FROM emp WHERE ename BETWEEN 'B' AND 'J';
SELECT * FROM emp WHERE ename BETWEEN 'B' AND 'K' AND ename!='K';

--display all emps with sal who are not in range of 1000 to 2000
SELECT * FROM emp;
SELECT * FROM emp WHERE sal>=1000;
SELECT * FROM emp WHERE sal<=2000;
SELECT * FROM emp WHERE NOT sal BETWEEN 1000 AND 2000;

--display all emps between B to J and T to Z
SELECT * FROM emp;
```

```
SELECT * FROM emp WHERE ename BETWEEN 'B' AND 'K' AND ename!='K';
SELECT * FROM emp WHERE ename>='T';

SELECT * FROM emp WHERE ename BETWEEN 'B' AND 'K' AND ename!='K' OR ename>='T';
```

## LIKE

```
- We have 2 wildcard characters which we can use with LIKE Operator
- % -> Any no of characters or even Empty
- _ -> Single Occurance of character
```

```
--dispaly all emps with name starting with M
SELECT * FROM emp;
SELECT * FROM emp WHERE ename>='M' AND ename<'N';
SELECT * FROM emp WHERE ename LIKE 'M';
SELECT * FROM emp WHERE ename LIKE 'M%';

--dispaly all emps with name starting with H
SELECT * FROM emp WHERE ename LIKE 'H%';

--dispaly all emps with name ending with H
SELECT * FROM emp WHERE ename LIKE '%H';

--diplay all emps having 'U' in their name
SELECT * FROM emp WHERE ename LIKE '%U%';

--display all emps having 'A' twice
SELECT * FROM emp WHERE ename LIKE '%A%A%';

--display all emps between B to J
SELECT * FROM emp WHERE ename BETWEEN 'B' AND 'J';
SELECT * FROM emp WHERE ename LIKE 'J%';
SELECT * FROM emp WHERE ename BETWEEN 'B' AND 'J' OR ename LIKE 'J%';

--display all emps with 4 letter name
SELECT * FROM emp WHERE ename LIKE '____';

--dispaly all emps having 'R' as the 3rd letter in their name
SELECT * FROM emp WHERE ename LIKE '__R%';

--display emps with 4 letter word with 'R' in the 3rd Position
SELECT * FROM emp WHERE ename LIKE '__R_';
```

## Practice Examples

```
--display emp highest salary in range of 1000 to 2000
SELECT * FROM emp;
SELECT * FROM emp WHERE sal BETWEEN 1000 AND 2000;
SELECT * FROM emp WHERE sal BETWEEN 1000 AND 2000 ORDER BY sal DESC;
SELECT * FROM emp WHERE sal BETWEEN 1000 AND 2000 ORDER BY sal DESC LIMIT 1;

--display clerk with min sal
SELECT * FROM emp WHERE job='CLERK';
SELECT * FROM emp WHERE job='CLERK' ORDER BY sal;
SELECT * FROM emp WHERE job='CLERK' ORDER BY sal LIMIT 1;

--display fifth lowest salary from dept 20 and 30.
SELECT * FROM emp WHERE deptno IN(20,30);
SELECT * FROM emp WHERE deptno IN(20,30) ORDER BY sal;
SELECT * FROM emp WHERE deptno IN(20,30) ORDER BY sal LIMIT 4,1;
SELECT * FROM emp WHERE deptno IN(20,30) ORDER BY sal LIMIT 4,1;
SELECT DISTINCT sal FROM emp WHERE deptno IN(20,30) ORDER BY sal LIMIT 4,1;
```

## DML

```
UPDATE emp SET empno=1000 WHERE ename='B';
UPDATE emp SET empno=1001 WHERE ename='J';
UPDATE emp SET empno=1003 WHERE ename='K';

--give sal hike of 200 to all emps who work as clerk
UPDATE emp SET sal=sal+200 WHERE job='CLERK';

--delete emps with empno as 1001,1000,1003
DELETE FROM emp WHERE empno=1001;
DELETE FROM emp WHERE empno=1000;
DELETE FROM emp WHERE empno=1003;

--delete all clerks
DELETE FROM emp WHERE job='CLERK';
```

## DDL

```
--If you want to delete all the data from a table then use Truncate
TRUNCATE emp;

-- If you want to remove entire table along with its structure the use drop
DROP TABLE emp;
```

- DELETE
    - It will delete the rows which matches with the condition
    - You can even delete all data from table without giving condition

    - This can be rolledback

- TRUNCATE
    - It will delete the entire data from the table
    - the table structure will remain as it is.
    - This cannot be rolledback

- DROP
    - It will remove the entire table from the databases.
    - This cannot be rolledback

## DUAL

```
- It is a single row,single column virtual table
```

```
SELECT * FROM books;
SELECT DATABASE();
SELECT DATABASE() FROM DUAL;

SELECT 345*345;
SELECT 345*345 FROM DUAL;

SELECT "HELLO WORLD" AS hello;
SELECT "HELLO WORLD"AS hello FROM DUAL;
```

## SQL Functions

```
- HELP FUNCTIONS
```

1. String Funcntion

```
HELP String Functions;

SELECT UPPER('sunbeam'); --SUNBEAM
SELECT LOWER('SUNBEAM'); --sunbeam

SELECT LOWER(ename) FROM emp;

SELECT LEFT('sunbeam',2);-- su
SELECT RIGHT('sunbeam',3); -- eam

SELECT SUBSTRING('SunBeam',3);
SELECT SUBSTRING('SunBeam',2,3);
```

```
SELECT SUBSTRING('SunBeam',-5);
SELECT SUBSTRING('SunBeam',-5,2);

--display emp-job like this -> SMITH-ANALYST
SELECT CONCAT(ename,'-',job) FROM emp;
SELECT CONCAT(ename,'-',job) AS emp_job FROM emp;

--display o/P as -> smith is working as ANALYST
SELECT CONCAT(ename,' is working as ',job) as emp_job FROM emp;
SELECT CONCAT(LOWER(ename),' is working as ',job) as emp_job FROM emp;

--display 1st letter of emp as upper and rest all as lower
SELECT UPPER(LEFT(ename,1)) FROM emp;
SELECT LOWER(SUBSTRING(ename,2)) FROM emp;
SELECT CONCAT(UPPER(LEFT(ename,1)),LOWER(SUBSTRING(ename,2))) AS ename FROM emp;

SELECT LENGTH('sunbeam');

SELECT TRIM('    sunbeam  ');

SELECT LENGTH(TRIM('    sunbeam  '));

SELECT LPAD('9388',10,'X');

SELECT RPAD('9388',10,'X');

--Homework -> 1234 XXXX XXXX 4567
```

2. Numeric Functions

```
HELP Numeric Functions;

SELECT POW(5,3);

SELECT SQRT(25);

SELECT ROUND(123.45);
SELECT ROUND(123.55,1);
SELECT ROUND(123.55,2);
SELECT ROUND(123.45,-1);
SELECT ROUND(123.45,-2);
SELECT ROUND(167.45,-1);

SELECT CEIL(58.25);
SELECT FLOOR(58.25);
```

3. DATE and TIME Functions

```
HELP Date and Time Functions

SELECT NOW();
SELECT SYSDATE();

SELECT NOW(),SLEEP(2),NOW();
--NOW() is going to give the time when quesy gets execute

SELECT SYSDATE(),SLEEP(2),SYSDATE();
--SYSDATE() is going to give the time when the function gets called.

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

SELECT DATE_ADD(NOW(),INTERVAL 84 DAY);

SELECT DATE_ADD(NOW(),INTERVAL 1 MONTH);

SELECT DATE_ADD(NOW(),INTERVAL 1 YEAR);

--to display experiance in no of days as o/p
SELECT ename,hire,DATEDIFF(NOW(),hire) FROM emp;

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

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

--display emp,hire,experiance  in terms of total year and months
SELECT ename,hire,TIMESTAMPDIFF(YEAR,hire,NOW()) AS
YEAR,TIMESTAMPDIFF(MONTH,hire,NOW())%12 AS MONTH FROM emp;

SELECT DAY(NOW()),MONTH(NOW()),YEAR(NOW());
SELECT HOUR(NOW()),MINUTE(NOW()),SECOND(NOW());

--display emps hired in 1981
SELECT * FROM emp WHERE YEAR(hire) = 1981;
```

# Control and NULL and List functions

- NULL is special value in RDBMS that represents absence of value in that column.
- NULL values do not work with relational operators and need to use special operators.
- Most of functions return NULL if NULL value is passed as one of its argument.
- IFNULL()
- NULLIF()

- GREATEST(), LEAST()

- IF(condition, true-value, false-value)

# Group functions

- Work on group of rows of table.
- Input to function is data from multiple rows & then output is single row. Hence these functions are called as "Multi Row Function“ or "Group Functions“.
- These functions are used to perform aggregate ops like sum, avg, max, min, count or std dev, etc. Hence these fns are also called as "Aggregate Functions".
- Example: SUM(), AVG(), MAX(), MIN(), COUNT().
- NULL values are ignored by group functions.
- Limitations of GROUP functions:
  - Cannot select group function along with a column.
  - Cannot select group function along with a single row fn.
  - Cannot use group function in WHERE clause/condition.
  - Cannot nest a group function in another group fn.

# GROUP BY clause

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

# Agenda

```
- Flow Control Functions
- Group Functions
- Group By
- Having Clause
- Joins - Relationships
```

4. Flow Control Functions

```
 HELP FLOW CONTROL FUNCTIONS

--display ename,deptname from emp.
-- 10 | ACCOUNTING, 20 | RESEARCH
SELECT ename,deptno FROM emp;

SELECT ename,deptno,CASE
WHEN deptno=10 THEN 'ACCOUNTING'
WHEN deptno=20 THEN 'RESEARCH'
ELSE 'OTHER'
END AS dname
FROM emp;

--dipaly emp as RICH if sal is more that 2500 and POOR if it is less than it.
SELECT ename,sal FROM emp;
SELECT ename,sal,IF(sal>2500,'RICH','POOR') AS Category FROM emp;

SELECT ename,sal,comm FROM emp;
SELECT ename,sal,comm,sal+comm AS Total_Income FROM emp;
SELECT ename,sal,comm,IFNULL(comm,0),sal+IFNULL(comm,0) AS Total_Income FROM emp;

--dispaly tax as null is sal is 800
SELECT ename,sal FROM emp;
SELECT ename,sal,NULLIF(sal,800) AS tax_income FROM emp;
```

# LIST Functions

```
SELECT CONCAT ('A','B','C','D',3.25);
SELECT CONCAT ('A','B',NULL,'D',3.25);
SELECT LEAST(100,200,50,25);
SELECT LEAST(100,NULL,50,25);
SELECT GREATEST(100,200,50,25);
SELECT GREATEST(100,NULL,50,25);
```

# Group Functions

```
- Single Row Function
    n input rows -> n output rows

- Group Function
    n input rows -> 1 output row
    Null values are ignored in Group functions
```

```
HELP Aggregate Functions and Modifiers

-- display empcount,total spending on sal, avg sal,min sal,max sal.

SELECT COUNT(empno),SUM(sal),AVG(sal),MAX(sal),MIN(sal) FROM emp;

SELECT COUNT(*),SUM(sal),AVG(sal),MAX(sal),MIN(sal) FROM emp;


SELECT COUNT(comm),SUM(comm),AVG(comm),MAX(comm),MIN(comm) FROM emp;
```

## Limitations of Group Functions

```
- SELECT @@sql_mode;
- Observe this -> ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION
- open notepad with administrator privileges
- Open my.ini file in your notepad.
- this file is available under C:\ProgramData\MySQL\MySQL Server 8.0
- Go under SERVER SECTION -> [mysqld] below this line you will find
- sql-mode="ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION"
- Restart your Mysql Server
```

```
-- display empname and avg sal.
SELECT ename,AVG(sal) FROM emp; --error
-- You cannot select any single column whenever you are using group functions

-- dispaly ename in lower case and avg sal
SELECT LOWER(ename),AVG(sal) FROM emp; --error'
--You cannot use single row functions whenever you are using group functions

--display all emps with Max sal -> MAX()
SELECT * FROM emp WHERE sal = MAX(sal);
--You cannot use group function in where clause

SELECT SUM(MAX(sal)) FROM emp;
--You cannot use nested group function
```

## GROUP BY CLAUSE

```
--dispaly dept wise total emp working in that dept and total sal spending
SELECT deptno,SUM(sal) AS Total_sal,COUNT(*) AS emp_count FROM emp; -- error

SELECT deptno,SUM(sal) AS Total_sal,COUNT(*) AS emp_count FROM emp GROUP BY
deptno;

SELECT deptno,SUM(sal) AS Total_sal,COUNT(*) AS emp_count FROM emp GROUP BY deptno
ORDER BY deptno;

--dispaly job wise total emp working in that job and total sal spending
SELECT * FROM emp ORDER BY job;
SELECT SUM(sal),COUNT(*) FROM emp GROUP BY job;
SELECT job,SUM(sal),COUNT(*) FROM emp GROUP BY job;

--diplay the deptno and the min sal of that dept
SELECT MIN(sal) FROM emp;
SELECT MIN(sal) FROM emp GROUP BY deptno;
SELECT deptno,MIN(sal) FROM emp GROUP BY deptno;

--diplay the job and the min sal for that job
SELECT MIN(sal) FROM emp;
SELECT MIN(sal) FROM emp GROUP BY job;
SELECT job,MIN(sal) FROM emp GROUP BY job;

--display count of emp working in dept with sepecific job
SELECT COUNT(*) FROM emp;
SELECT COUNT(*) FROM emp GROUP BY deptno;
SELECT COUNT(*) FROM emp GROUP BY job;
SELECT COUNT(*) FROM emp GROUP BY deptno,job;
SELECT deptno,COUNT(*) FROM emp GROUP BY deptno,job;
SELECT deptno,job,COUNT(*) FROM emp GROUP BY deptno,job;

SELECT deptno,job,COUNT(*) FROM emp GROUP BY deptno,job ORDER BY deptno;
```

## Having Clause

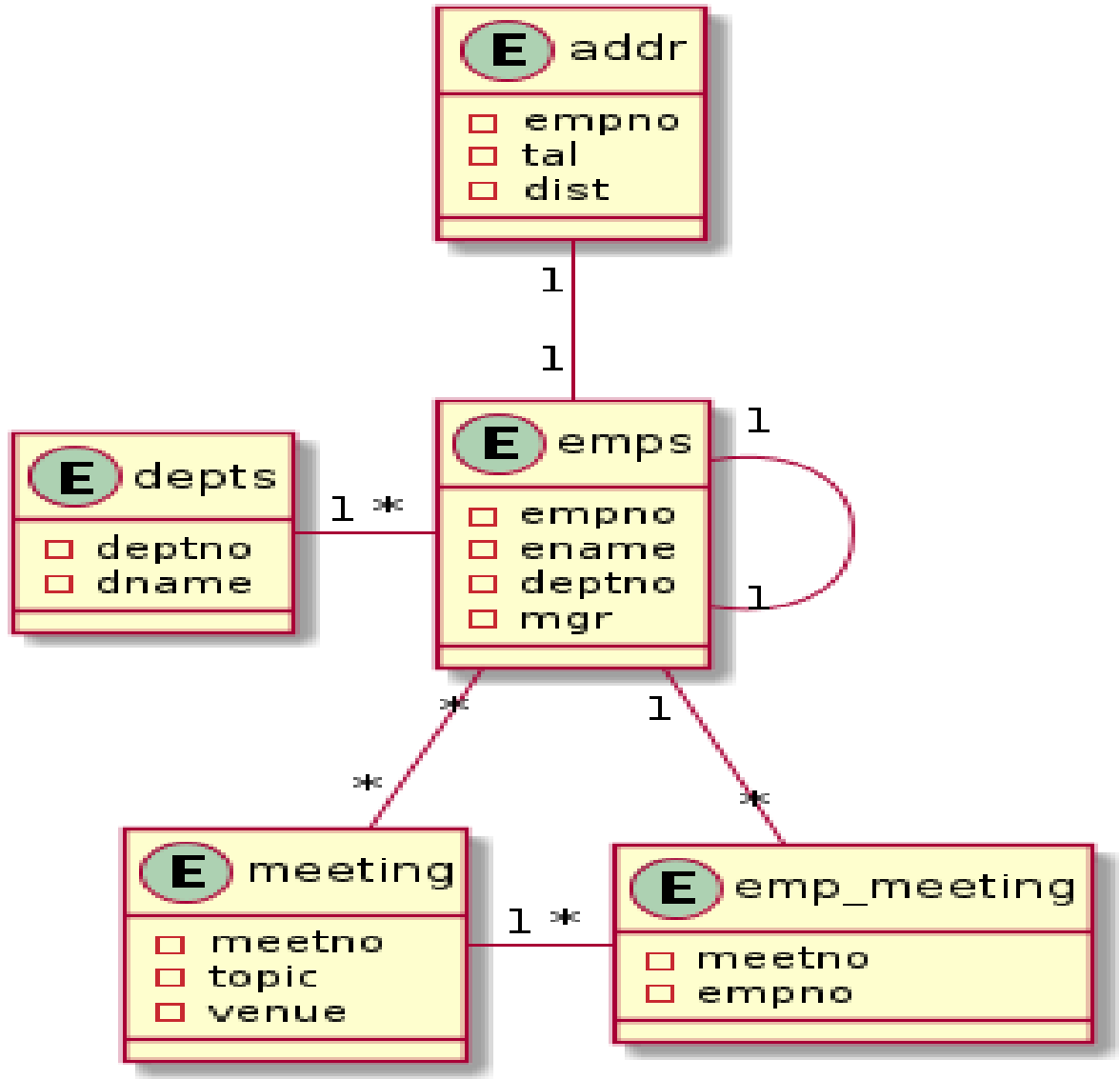
```
- Having clause must be used only with group by clause
- If their is condition on aggregate/group values
```

```
-- display deptwise total salary spend if total sal > 9000
SELECT deptno,SUM(sal) FROM emp;
SELECT deptno,SUM(sal) FROM emp GROUP BY deptno;
SELECT deptno,SUM(sal) FROM emp GROUP BY deptno HAVING SUM(sal)>9000;
```

```sql
--display job wise avg sal where the avg sal is > 2500
SELECT job,AVG(sal) FROM emp GROUP BY job;
SELECT job,AVG(sal) FROM emp GROUP BY job HAVING AVG(sal)>2500;

--display max sal of all jobs of dept 10 and 20
SELECT deptno,job,MAX(sal) FROM emp GROUP BY deptno,job;
SELECT deptno,job,MAX(sal) FROM emp GROUP BY deptno,job HAVING deptno IN (10,20);
SELECT deptno,job,MAX(sal) FROM emp WHERE deptno IN(10,20) GROUP BY deptno,job;

--display max sal of all jobs of dept 10 and 20 where max sal > 2500
SELECT deptno,job,MAX(sal) FROM emp
WHERE deptno IN(10,20)
GROUP BY deptno,job
HAVING MAX(sal)>2500;

--dispaly only one dept that spends maximum on emps salary.
SELECT deptno,SUM(sal) FROM emp GROUP BY deptno;
SELECT deptno,SUM(sal) FROM emp GROUP BY deptno ORDER BY SUM(sal) DESC;
SELECT deptno,SUM(sal) FROM emp GROUP BY deptno ORDER BY SUM(sal) DESC LIMIT 1;

--dispaly job having lowest avg(sal)
SELECT job,AVG(sal) FROM emp GROUP BY job;
SELECT job,AVG(sal) FROM emp GROUP BY job ORDER BY AVG(sal);
SELECT job,AVG(sal) FROM emp GROUP BY job ORDER BY AVG(sal) LIMIT 1;

--display job having lowest avg income
SELECT job,AVG(sal+IFNULL(comm,0)) AS income FROM emp
GROUP BY job
ORDER BY income
LIMIT 1;
```

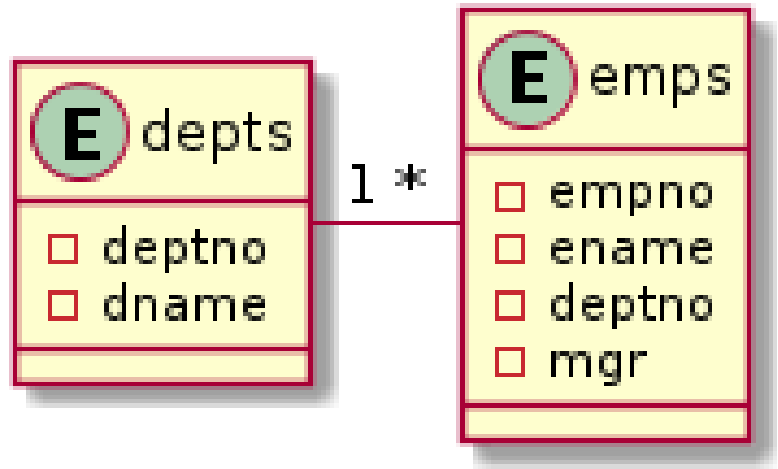
# Entity Relations

- To avoid redundancy of the data, data should be organized into multiple tables so that tables are related to each other.
- The relations can be one of the following
  - One to One
  - One to Many
  - Many to One
  - Many to Many
- Entity relations is outcome of Normalization process.



# SQL Joins

- Join statements are used to SELECT data from multiple tables using single query.
- Typical RDBMS supports following types of joins:
  - Cross Join
  - Inner Join
  - Left Outer Join
  - Right Outer Join
  - Full Outer Join
  - Self join



# Cross Join

| deptno | dname |
|---|---|
| 10 | DEV |
| 20 | QA |
| 30 | OPS |
| 40 | ACC |

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

- Compares each row of Table1 with every row of Table2.
- Yields all possible combinations of Table1 and Table2.
- In MySQL, The larger table is referred as "Driving Table", while smaller table is referred as "Driven Table". Each row of Driving table is combined with every row of Driven table.
- Cross join is the fastest join, because there is no condition check involved.

# Inner Join

| deptno | dname |
|---|---|
| 10 | DEV |
| 20 | QA |
| 30 | OPS |
| 40 | ACC |

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

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

# Left Outer Join

| deptno | dname |
|---|---|
| 10 | DEV |
| 20 | QA |
| 30 | OPS |
| 40 | ACC |

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

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

# Right Outer Join

| deptno | dname |
|---|---|
| 10 | DEV |
| 20 | QA |
| 30 | OPS |
| 40 | ACC |

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

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

# Full Outer Join

| deptno | dname |
|---|---|
| 10 | DEV |
| 20 | QA |
| 30 | OPS |
| 40 | ACC |

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

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

# Set operators

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

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

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

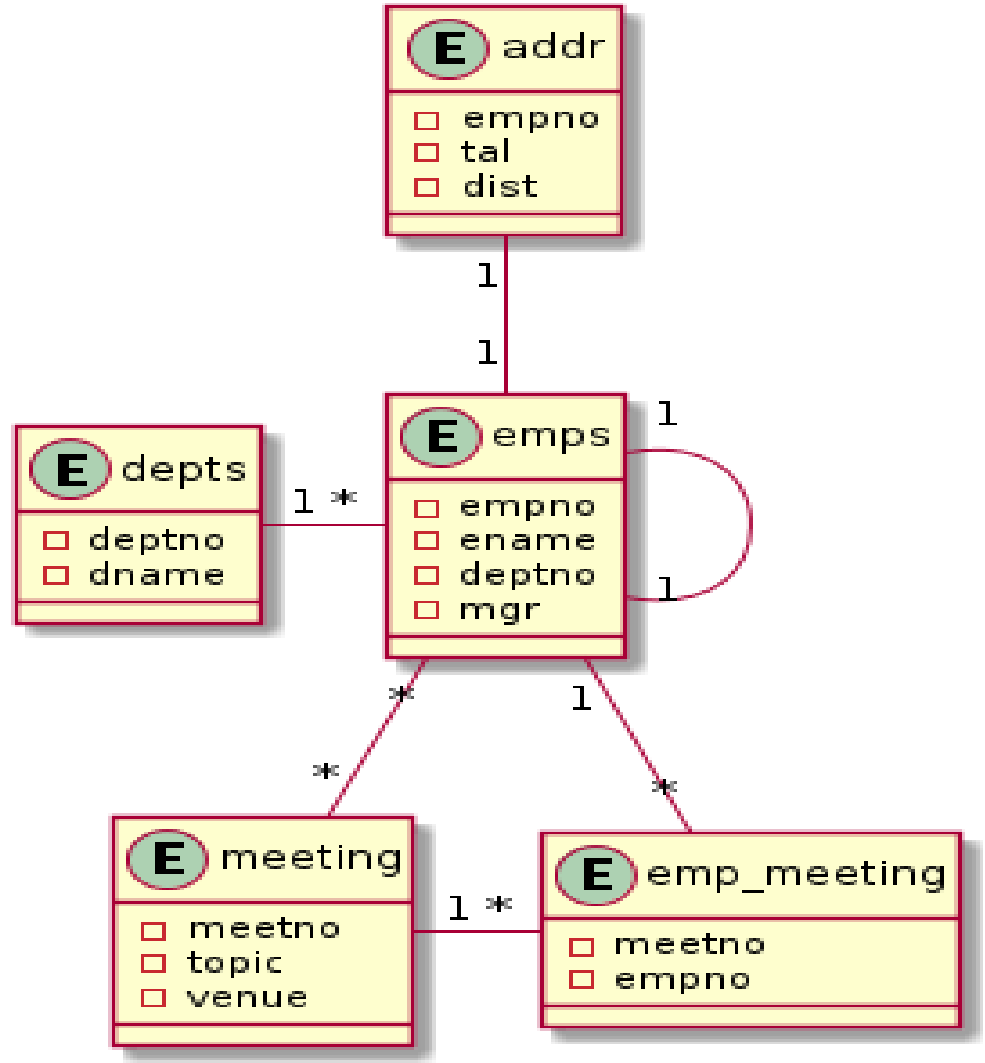
# Self Join

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

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

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

# Multi-Table Joins



# Agenda

```
-JOINS
```

# JOINS

1. CROSS JOIN

```
--display ename,deptname from emps and depts
SELECT e.ename,d.dname FROM emps e CROSS JOIN depts d;

SELECT ename,dname FROM emps CROSS JOIN depts;

SELECT ename,deptno,dname FROM emps CROSS JOIN depts; --error

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

2. INNER JOIN

```
--display ename,deptname from emps and depts
SELECT e.ename,d.dname FROM emps e INNER JOIN depts d ON e.deptno=d.deptno;

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

SELECT e.ename,d.dname FROM emps e INNER JOIN depts d ON e.deptno!=d.deptno;
--non-equi joins
```

3. LEFT OUTER JOIN
    - LEFT OUTER JOIN = INTERESECTION + DATA FROM LEFT TABLE

```
--display ename,deptname from emps and depts
SELECT e.ename,d.dname FROM emps e LEFT OUTER JOIN depts d ON e.deptno=d.deptno;

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

4. RIGHT OUTER JOIN
    - RIGHT OUTER JOIN = INTERESECTION + DATA FROM RIGHT TABLE

```
--display ename,deptname from emps and depts
SELECT e.ename,d.dname FROM emps e RIGHT OUTER JOIN depts d ON e.deptno=d.deptno;
```

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

5. FULL OUTER JOIN
    - FULL OUTER JOIN = LEFT OUTER JOIN + RIGHT OUTER JOIN
    - This is not supported in mysql
    - But however this can be implemented by using set operators

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

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

SELECT e.ename,d.dname FROM emps e LEFT OUTER JOIN depts d ON e.deptno=d.deptno
UNION ALL
SELECT e.ename,d.dname FROM emps e RIGHT OUTER JOIN depts d ON e.deptno=d.deptno;
```

6. SELF JOIN

```
--display ename and manager name of that emp.
SELECT e.ename,m.ename AS mname FROM emps e INNER JOIN emps m ON e.mgr=m.mgr;
--wrong o/p

SELECT e.ename,m.ename AS mname FROM emps e INNER JOIN emps m ON e.mgr=m.empno;

SELECT e.ename,m.ename AS mname FROM emps e LEFT JOIN emps m ON e.mgr=m.empno;
```

# JOINS PRACTICE

```
SELECT * FROM emps;
SELECT * FROM depts;
SELECT * FROM meeting;
SELECT * FROM addr;
SELECT * FROM emp_meeting;

--display empno,ename,deptno and deptname of all employees
SELECT e.empno,e.ename,e.deptno,d.dname FROM emps e
INNER JOIN depts d ON e.deptno=d.deptno;

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

```sql
--display ename,deptname and address of that employee
SELECT e.ename,d.dname FROM emps e
INNER JOIN depts d ON e.deptno=d.deptno;

SELECT e.ename,a.tal,a.dist FROM emps e
INNER JOIN addr a ON e.empno=a.empno;

SELECT e.ename,a.tal,a.dist,d.dname FROM emps e
INNER JOIN addr a ON e.empno=a.empno
INNER JOIN depts d ON e.deptno=d.deptno;

SELECT e.ename,d.dname,a.tal,a.dist FROM emps e
INNER JOIN addr a ON e.empno=a.empno
LEFT JOIN depts d ON e.deptno=d.deptno;

--display emp and their meeting topics
SELECT * FROM emp;
SELECT * FROM meeting;
SELECT * FROM emp_meeting;

SELECT e.ename,em.meetno FROM emps e
INNER JOIN emp_meeting em ON e.empno=em.empno;

SELECT m.topic,em.empno FROM meeting m
INNER JOIN emp_meeting em ON m.meetno=em.meetno;

SELECT e.ename,m.topic FROM emps e
INNER JOIN emp_meeting em ON e.empno=em.empno
INNER JOIN meeting m ON m.meetno=em.meetno;

-- display empname,meeting topic and his address
SELECT e.ename,m.topic FROM emps e
INNER JOIN emp_meeting em ON e.empno=em.empno
INNER JOIN meeting m ON m.meetno=em.meetno;

SELECT e.ename,a.tal,a.dist FROM emps e
INNER JOIN addr a ON e.empno=a.empno;

SELECT e.ename,m.topic,a.tal,a.dist FROM emps e
INNER JOIN emp_meeting em ON e.empno=em.empno
INNER JOIN meeting m ON m.meetno=em.meetno
INNER JOIN addr a ON e.empno=a.empno;

-- display empname,deptname,meeting topic and his address
SELECT e.ename,m.topic,a.tal,a.dist FROM emps e
INNER JOIN emp_meeting em ON e.empno=em.empno
INNER JOIN meeting m ON m.meetno=em.meetno
INNER JOIN addr a ON e.empno=a.empno;

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

SELECT e.ename,d.dname,m.topic,a.tal,a.dist FROM emps e
INNER JOIN emp_meeting em ON e.empno=em.empno
```

```
INNER JOIN meeting m ON m.meetno=em.meetno
INNER JOIN addr a ON e.empno=a.empno
LEFT JOIN depts d ON e.deptno=d.deptno;

--print deptname and count of employees in that dept
SELECT * FROM emps;
SELECT deptno,COUNT(empno) FROM emps GROUP BY deptno;

SELECT d.dname,COUNT(e.empno) FROM emps e
INNER JOIN depts d ON e.deptno=d.deptno
GROUP BY d.dname;

SELECT d.dname,COUNT(e.empno) FROM emps e
RIGHT JOIN depts d ON e.deptno=d.deptno
GROUP BY d.dname;

--display emps and their total meetings in desc order of their meeting count
SELECT e.ename,em.meetno FROM emps e
INNER JOIN emp_meeting em ON e.empno=em.empno;

SELECT e.ename,COUNT(em.meetno) AS emp_count FROM emps e
INNER JOIN emp_meeting em ON e.empno=em.empno
GROUP BY e.ename
ORDER BY emp_count DESC;

--display all emps from 'dev' dept
SELECT e.ename,d.dname FROM emps e
INNER JOIN depts d ON e.deptno=d.deptno
WHERE d.dname='DEV';
```

## Usage of all Clauses as per priority

```
SELECT columns FROM table1
XXX JOIN table2 XXX ON condition
XXX JOIN table3 XXX ON condition
WHERE condition
GROUP BY column
Having condition
ORDER BY column
LIMIT n
```

## Non standard joins

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

SELECT e.ename,d.dname FROM emps e
```

```
JOIN depts d ON e.deptno=d.deptno;

SELECT e.ename,d.dname FROM emps e
CROSS JOIN depts d ON e.deptno=d.deptno;

SELECT e.ename,d.dname FROM emps e
CROSS JOIN depts d WHERE e.deptno=d.deptno;

SELECT e.ename,d.dname FROM emps e,
depts d WHERE e.deptno=d.deptno;

SELECT e.ename,d.dname FROM emps e
INNER JOIN depts d USING(deptno);
--this join only works in mysql

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

# Transaction

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

# Transaction

- Transaction management
  - START TRANSACTION;
  - …
  - COMMIT WORK;

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

# Transaction

- Save-point is state of database tables (data) at the moment (within a transaction).
- It is advised to create save-points at end of each logical section of work.
- Database user may choose to rollback to any of the save-point.
- Transaction management with Save-points
  - START TRANSACTION;
  - …
  - SAVEPOINT sa1;
  - …
  - SAVEPOINT sa2;
  - …
  - ROLLBACK TO sa1;
  - …
  - COMMIT; // or ROLLBACK
- Commit always commit the whole transaction.
- ROLLBACK or COMMIT clears all save-points.

# Transaction

- Transaction is set of DML statements.
- If any DDL statement is executed, current transaction is automatically committed.
- Any power failure, system or network failure automatically rollback current state.
- Transactions are isolated from each other and are consistent.

# Row locking

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

# Row locking

- Manually locking the row in advanced before issuing UPDATE or DELETE is known as "PESSIMISTIC LOCKING".
- This is done by appending FOR UPDATE to the SELECT query.
- It will lock all selected rows, until transaction is committed or rollbacked.
- If these rows are already locked by another users, the SELECT operation is blocked until rows lock is released.

- By default MySQL does table locking. Row locking is possible only when table is indexed on the column.

# Data Control Language

- Security is built-in feature of any RDBMS. It is implemented in terms of permissions (a.k.a. privileges).
- There are two types of privileges.
- System privileges
  - Privileges for certain commands i.e. CREATE, ALTER, DROP, ...
  - -Typically these privileges are given to the database administrator or higher authority user.
- Object privileges
  - RDBMS objects are table, view, stored procedure, function, triggers, …
  - -Can perform operations on the objects i.e. INSERT, UPDATE, DELETE, SELECT, CALL, ...
  - Typically these privileges are given to the database users.

# User Management

- User management is responsibility of admin (root).
- New user can be created using CREATE USER.
  - CREATE USER user@host IDENTIFIED BY 'password';
  - host can be hostname of server, localhost (current system) or '%' for all client systems.
- Permissions for the user can be listed using SHOW GRANTS command.
  - SHOW GRANTS FOR user@host;
- Users can be deleted using DROP USER.
  - DROP USER user@host;
- Change user password.
  - ALTER USER user@host IDENTIFIED BY 'new_password';
  - FLUSH PRIVILEGES;

# Data Control Language

- Permissions are given to user using GRANT command.
  - GRANT CREATE ON db.* TO user@host;
  - GRANT CREATE ON *.* TO user1@host, user2@host;
  - GRANT SELECT ON db.table TO user@host;
  - GRANT SELECT, INSERT, UPDATE ON db.table TO user@host;
  - GRANT ALL ON db.* TO user@host;
- By default one user cannot give permissions to other user. This can be enabled using WITH GRANT OPTION.
  - GRANT ALL ON *.* TO user@host WITH GRANT OPTION;
- Permissions assigned to any user can be withdrawn using REVOKE command.
  - REVOKE SELECT, INSERT ON db.table FROM user@host;
- Permissions can be activated by FLUSH PRIVILEGES.
  - System GRANT tables are reloaded by this command. Auto done after GRANT, REVOKE.
  - Command is necessary is GRANT tables are modified using DML operations.

# Agenda

```
Security
Transactions
Row Locking
```

## Security

```
mysql> PROMPT \u>

root>SELECT user FROM mysql.user;

root>CREATE USER mgr IDENTIFIED BY 'mgr';

root>SELECT user FROM mysql.user;

root>SHOW DATABASES;

--to give permissions to mgr on specific table
root>GRANT ALL PRIVILEGES ON classwork.emp TO mgr;

--open new cmd prompt
cmd>mysql -u mgr -pmgr

mgr>SHOW DATABASES;

mgr>SHOW TABLES;

root>CREATE USER teamlead@localhost IDENTIFIED BY 'team';

root>SELECT user,host FROM mysql.user;

--open new cmd prompt
cmd>mysql -u teamlead -pteam

teamlead>SHOW DATABASES;

mgr>SHOW GRANTS;

root>SHOW GRANTS FOR mgr;

root>SHOW GRANTS FOR teamlead@localhost;

mgr>GRANT ALL PRIVILEGES ON classwork.emp TO teamlead@localhost; -- error

root>GRANT ALL PRIVILEGES ON classwork.* TO mgr WITH GRANT OPTION;

mgr>GRANT ALL PRIVILEGES ON classwork.emp TO teamlead@localhost; --OK
```

```
mgr>GRANT ALL PRIVILEGES ON classwork.dept TO teamlead@localhost;

root>CREATE USER dev1 IDENTIFIED BY 'dev1';

--open new cmd prompt
cmd>mysql -u dev1 -pdev1

mgr>GRANT SELECT ON classwork.emp TO dev1;

mgr>GRANT INSERT,UPDATE ON classwork.emp TO dev1;

mgr>REVOKE UPDATE ON classwork.emp FROM dev1;

root>DROP USER dev1;
```

## Transactions - TCL

```
TCL COMMANDS
    - START TRANSACTION
    - ROLLBACK
    - COMMIT
```

```
CREATE TABLE accounts(
    id INT,
    type CHAR(10),
    balance DECIMAL(9,2)
);

INSERT INTO accounts VALUES
(1,'SAVINGS',2000),
(2,'CURRENT',3000),
(3,'SAVINGS',4000),
(4,'CURRENT',5000),
(5,'SAVINGS',6000);

SELECT * FROM accounts;

root>UPDATE accounts set balance=balance-1000 WHERE id=5;
root>SELECT * FROM accounts;
mgr>SELECT * FROM accounts;

root>START TRANSACTION
root>UPDATE accounts set balance=balance-1000 WHERE id=4;
root>SELECT * FROM accounts;
mgr>SELECT * FROM accounts;
root>ROLLBACK;
root>SELECT * FROM accounts;
mgr>SELECT * FROM accounts;
```

```
root>START TRANSACTION
root>UPDATE accounts set balance=balance-1000 WHERE id=4;
root>SELECT * FROM accounts;
mgr>SELECT * FROM accounts;
root>COMMIT;
root>SELECT * FROM accounts;
mgr>SELECT * FROM accounts;
```

## SAVEPOINT

```
- Save point can be called as a state in a transaction
```

```
root>START TRANSACTION;
root>INSERT INTO accounts VALUES(6,"TEMP",1000);
root>SELECT * FROM acounts;
root>SAVEPOINT sp1;

root>DELETE FROM accounts WHERE id=4;
root>DELETE FROM accounts WHERE id=5;
root>SELECT * FROM acounts;
root>SAVEPOINT sp2;

root>UPDATE accounts SET balance=balance+2000 WHERE id=1;
root>SELECT * FROM acounts;

root>ROLLBACK TO sp2;
root>SELECT * FROM accounts;

root>ROLLBACK;
root>SELECT * FROM accounts;
```

## Transaction Properties

```
A - Atomicity
    DML opertaions will either be successfull/Failure. Partial transactions are
never commited.
C - Consistency
    At the end of transaction same state will be visible to all the users.
I - Isolation
    Every transaction is isolated from each other.
D - Durability
    At the end of transaction the state will be saved on to the server.
```

## ROW LOCKING

```
root>START TRANSACTION
root>UPDATE accounts set balance=balance-5000 WHERE id=5;
root>SELECT * FROM accounts;
mgr>SELECT * FROM accounts;
mgr>UPDATE accounts set balance=balance-5000 WHERE id=5; --terminal hangs
root>ROLLBACK;
root>SELECT * FROM accounts;
mgr>SELECT * FROM accounts;
```

## TABLE LOCKING

```
If your table does not have a primary key then your whole table gets locked.
```

```
root>START TRANSACTION
root>UPDATE accounts set balance=balance-4000 WHERE id=4;
root>SELECT * FROM accounts;
mgr>SELECT * FROM accounts;
mgr>UPDATE accounts set balance=balance-4000 WHERE id=4; --terminal hangs
mgr>UPDATE accounts set balance=balance-3000 WHERE id=2; --terminal hangs

root>SELECT * FROM accounts;
mgr>SELECT * FROM accounts;
```

## Pessimistic Locking

```
root>START TRANSACTION;
root>SELECT id,name,subject FROM books WHERE id=2002 FOR UPDATE;

mgr>UPDATE books set subject='D++ Programming' WHERE id=2002;--terminal hangs
```

# Sub queries

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

# Sub queries

- Multi-row sub-query
  - Sub-query returns multiple rows.
  - Usually it is compared in outer query using operators like IN, ANY or ALL.
    - IN operator compare for equality with results from sub-queries (at least one result should match)..
    - ANY operator compares with the results from sub-queries (at least one result should match).
    - ALL operator compares with the results from sub-queries (all results should match).

# Sub queries

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

# Sub query

- Sub queries with UPDATE and DELETE are not supported in all RDBMS.
- -In MySQL, Sub--queries in UPDATE/DELETE is allowed, but sub--query should not SELECT from the same table, on which UPDATE/DELETE operation is in progress.

# Agenda

```
SubQuery
```

# SubQuery

```
- It is query inside another query
1. Single Row Subquery
2. Multi Row Subquery
```

1. Single Row Subquery

```
--display all emps with maximum sal.
SELECT max(sal) FROM emp;
SELECT * FROM emp ORDER BY sal DESC LIMIT 1;

SELECT * FROM emp WHERE sal=MAX(sal); -- error
SELECT max(sal) FROM emp;
SELECT * FROM emp WHERE sal=5000;

SELECT * FROM emp WHERE sal=(SELECT max(sal) FROM emp);


--display all emps with second highest salary
SELECT DISTINCT sal FROM emp ORDER BY sal DESC LIMIT 1,1;
SELECT * FROM emp WHERE sal=(SELECT DISTINCT sal FROM emp ORDER BY sal DESC LIMIT
1,1);

--display all emps working in dept same as employee KING
SELECT deptno FROM emp WHERE ename='KING';
SELECT * FROM emp WHERE deptno=(SELECT deptno FROM emp WHERE ename='KING');
SELECT * FROM emp WHERE deptno=(SELECT deptno FROM emp WHERE ename='KING') AND
ename<>'KING';

--display all emps working in job same as that of SCOTT
SELECT job FROM emp WHERE ename='SCOTT';
SELECT * FROM emp WHERE job = (SELECT job FROM emp WHERE ename='SCOTT');
SELECT * FROM emp WHERE job = (SELECT job FROM emp WHERE ename='SCOTT') AND
ename!='SCOTT';
```

2. Multi Row Subquery
    - The inner query return multiple rows.
    - the multiple rows from inner query can be compared by using IN,ANY,ALL operators

```
-- display all emps having sal more than all salesman
SELECT sal FROM emp where job='SALESMAN';
SELECT * FROM emp WHERE sal > ALL(SELECT sal FROM emp where job='SALESMAN');
--(sal>1600 AND sal>1250 AND sal>1500)

-- display all emps having sal less than any dept of 20
SELECT sal FROM emp WHERE deptno=20;
SELECT * FROM emp WHERE sal< ANY(SELECT sal FROM emp WHERE deptno=20);
--(sal<800 OR sal<2975 OR sal< 3000 OR...)

--display all dept name which have employees
SELECT DISTINCT deptno FROM emp;
SELECT * FROM dept WHERE deptno= ANY(SELECT DISTINCT deptno FROM emp);

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

--display all dept name which dont have employees

SELECT * FROM dept WHERE deptno != ALL(SELECT DISTINCT deptno FROM emp);
SELECT * FROM dept WHERE deptno NOT IN(SELECT DISTINCT deptno FROM emp);
```

## ALL, ANY vs IN Operator

```
-ANY can be used only with subquery, In can be used with/without subquery
-ANY can be use with all relational operators,IN can be used with only for equal
condition
-ANY behaves like logical OR
-ALL can be used only with sub query
-ALL can be used with all relational operators
-All behaves like logical AND
```

## Corelated Subquery

```
If your inner query is having a condition(Where clause) based on current row of
outer query then such type of query is called as corelated subquery
```

```
--display all dept name which have employees
SELECT * FROM dept WHERE deptno= ANY(SELECT deptno FROM emp);
--10 - ACC - SELECT deptno FROM emp;- 13 rows
--20 - RES - SELECT deptno FROM emp;- 13 rows
--30 - SAL - SELECT deptno FROM emp;- 13 rows
--40 - OPS - SELECT deptno FROM emp;- 13 rows

--display all dept name which have employees
SELECT * FROM dept WHERE deptno= ANY(SELECT DISTINCT deptno FROM emp);
```

```
--10 - ACC - SELECT DISTINCT deptno FROM emp;-10,20,30
--20 - RES - SELECT DISTINCT deptno FROM emp;-10,20,30
--30 - SAL - SELECT DISTINCT deptno FROM emp;-10,20,30
--40 - OPS - SELECT DISTINCT deptno FROM emp;-10,20,30

SELECT * FROM dept d WHERE d.deptno=ANY
(SELECT e.deptno FROM emp e WHERE e.deptno=d.deptno);
--10 - ACC - SELECT e.deptno FROM emp e WHERE e.deptno=d.deptno-10,10,10
--20 - RES - SELECT e.deptno FROM emp e WHERE e.deptno=d.deptno-20 ->5 rows
--30 - SAL - SELECT e.deptno FROM emp e WHERE e.deptno=d.deptno-30 -> 6 rows
--40 - OPS - SELECT e.deptno FROM emp e WHERE e.deptno=d.deptno-0 rows

SELECT * FROM dept d WHERE d.deptno=
(SELECT DISTINCT e.deptno FROM emp e WHERE e.deptno=d.deptno);
--10 - ACC - SELECT e.deptno FROM emp e WHERE e.deptno=d.deptno-10
--20 - RES - SELECT e.deptno FROM emp e WHERE e.deptno=d.deptno-20
--30 - SAL - SELECT e.deptno FROM emp e WHERE e.deptno=d.deptno-30
--40 - OPS - SELECT e.deptno FROM emp e WHERE e.deptno=d.deptno-0 rows
```

## Subquery in Projection

```
--display dept wise count of employees and total no of employees
--deptno - countof emp - total emp
--20          5                14
--30          6                14
--10          3                14

SELECT deptno,COUNT(empno) FROM emp;--error
SELECT deptno,COUNT(empno) FROM emp GROUP BY deptno;
SELECT COUNT(empno) FROM emp;

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

SELECT deptno,COUNT(empno) AS empcount,(SELECT COUNT(empno) FROM emp) AS
totalcount FROM emp GROUP BY deptno;
```

## Subquery in FROM clause

```
- If subquery is written in FROM clause then it should compulsary have an alias
- The subquery in from clause is called as derived table or inline view.
```

```
--diplay emp as rich is sal>2000 and poor if sal<2000
SELECT ename,sal,CASE
WHEN sal>=2000 THEN 'RICH'
WHEN sal<2000 THEN 'POOR'
```

```
END
AS category
FROM emp;

-- display count of employess in above category
SELECT category,COUNT(ename) FROM
(SELECT ename,sal,CASE
WHEN sal>=2000 THEN 'RICH'
WHEN sal<2000 THEN 'POOR'
END
AS category
FROM emp) AS category_count
GROUP BY category;
```

## Subquery in DML operations

```
--insert employee with dept as operations
SELECT deptno FROM dept WHERE dname='OPERATIONS';

INSERT INTO emp(empno,ename,sal,deptno) VALUES
(1001,'Rohan',1234,(SELECT deptno FROM dept WHERE dname='OPERATIONS'));


--update employee and change sal to 5678 with dept as operations
UPDATE emp set sal=5678 WHERE deptno=(SELECT deptno FROM dept WHERE
dname='OPERATIONS');

--delete all emps from operations department
DELETE FROM emp WHERE deptno=(SELECT deptno FROM dept WHERE dname='OPERATIONS');

--delete from emp having max sal.
DELETE FROM emp WHERE sal=(SELECT MAX(sal) FROM emp);
--error bcoz you cant select from same table on which you are performing DML
```

## SQL PERFORMANCE

```
--display all emps having sal < max of sal
SELECT * FROM emp WHERE sal < (SELECT max(sal) FROM emp);

--display all emps having sal < max of sal from dept 20
SELECT * FROM emp WHERE sal < (SELECT max(sal) FROM emp WHERE deptno=20);

SELECT * FROM emp WHERE sal < ANY(SELECT sal FROM emp WHERE deptno=20);

EXPLAIN FORMAT=JSON SELECT * FROM emp WHERE sal < ANY(SELECT sal FROM emp WHERE
deptno=20);

EXPLAIN FORMAT=JSON SELECT * FROM emp WHERE sal < (SELECT max(sal) FROM emp WHERE
deptno=20);
```

# Views

- RDBMS view represents view (projection) of the data.
- View is based on SELECT statement.
- Typically it is restricted view of the data (limited rows or columns) from one or more tables (joins and/or sub-queries) or summary of the data (grouping).
- Data of view is not stored on server hard-disk; but its SELECT statement is stored in compiled form. It speed up execution of view.

# Views

- Views are of two types: Simple view and Complex view
- Usually if view contains computed columns, group by, joins or sub-queries, then the views are said to be complex. DML operations are not supported on these views.
- DML operations on view affects underlying table.
- View can be created with CHECK OPTION to ensure that DML operations can be performed only the data visible in that view.

# View

- Views can be differentiated with: SHOW FULL TABLES.
- Views can be dropped with DROP VIEW statement.
- View can be based on another view.

- Applications of views
  - Security: Providing limited access to the data.
  - Hide source code of the table.
  - Simplifies complex queries.

# Index

- Index enable faster searching in tables by indexed columns.
  - CREATE INDEX idx_name ON table(column);
- One table can have multiple indexes on different columns/order.
- Typically indexes are stored as some data structure (like BTREE or HASH) on disk.
- Indexes are updated during DML operations. So DML operation are slower on indexed tables.

# Index

- Index can be ASC or DESC.
  - It cause storage of key values in respective order (MySQL 8.x onwards).
  - ASC/DESC index is used by optimizer on ORDER BY queries.
- Types of indexes:
  - Simple index
    - CREATE INDEX idx_name ON table(column [ASC|DESC]);
  - Unique index
    - CREATE UNIQUE INDEX idx_name ON table(column [ASC|DESC]);
    - Doesn’t allow duplicate values.
  - Clustered index
    - PRIMARY index automatically created on Primary key for row lookup.
    - If primary key is not available, hidden index is created on synthetic column.
    - It is maintained in tabular form and its reference is used in other indexes.

# Index

- Indexes should be created on shorter (INT, CHAR, …) columns to save disk space.
- Few RDBMS do not allow indexes on external columns i.e. TEXT, BLOB.
- MySQL support indexing on TEXT/BLOB up to n characters.
  - CREATE TABLE test (blob_col BLOB, …, INDEX(blob_col(10)));
- To list all indexes on table:
  - SHOW INDEXES FROM table;
- To drop an index:
  - DROP INDEX idx_name ON table;
- When table is dropped, all indexes are automatically dropped.
- Indexes should not be created on the columns not used frequent search, ordering or grouping operations.
- Columns in join operation should be indexed for better performance.

# Agenda

```
- Subquery
    EXISTS
    NOT EXISTS
- VIEWS
- INDEXES
```

```
--display dname in which employees exist
SELECT * FROM dept WHERE deptno =ANY (SELECT deptno FROM emp);
SELECT * FROM dept WHERE deptno =ANY (SELECT DISTINCT deptno FROM emp);
SELECT * FROM dept d WHERE deptno = (SELECT DISTINCT e.deptno FROM emp e WHERE
e.deptno=d.deptno);

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

--display dname in which employees does't exist
SELECT * FROM dept d WHERE NOT EXISTS (SELECT DISTINCT e.deptno FROM emp e WHERE
e.deptno=d.deptno);
```

# VIEWS

```
1. SIMPLE VIEW
    -DQL + DML
2. COMPLEX VIEW
    -DQL

View is a projection of data
```

```
--display emp,sal and their category
SELECT ename,sal,CASE
WHEN sal>2000 THEN 'RICH'
WHEN sal<=2000 THEN 'POOR'
END AS category
FROM emp;

-- create a view for above query
CREATE VIEW v_emp_category AS
SELECT ename,sal,CASE
WHEN sal>2000 THEN 'RICH'
WHEN sal<=2000 THEN 'POOR'
END AS category
FROM emp;
```

```
SHOW TABLES;

SHOW FULL TABLES;

DESC v_emp_category;

SELECT * FROM v_emp_category;

SELECT ename,category FROM v_emp_category;

SELECT category,count(ename) FROM v_emp_category GROUP BY category;

--add one more category as MIDDLE where 1000<sal<2000
SELECT ename,sal,CASE
WHEN sal>2000 THEN 'RICH'
WHEN sal>1000 AND sal<2000 THEN 'MIDDLE'
WHEN sal<=1000 THEN 'POOR'
END AS category
FROM emp;

--alter the v_emp_category view for above requirement
ALTER VIEW v_emp_category AS
SELECT ename,sal,CASE
WHEN sal>2000 THEN 'RICH'
WHEN sal>1000 AND sal<2000 THEN 'MIDDLE'
WHEN sal<=1000 THEN 'POOR'
END AS category
FROM emp;

SHOW TABLES;

SELECT * FROM v_emp_category;

SHOW CREATE VIEW v_emp_category;
--It will show how the view is created

--delete a view
DROP VIEW v_emp_category;
```

```
--create a view on empno,ename,sal
CREATE VIEW v_emp_sal AS
SELECT empno,ename,sal FROM emp;

--create a view on empno,ename,sal where sal>2500
CREATE VIEW v_richemp AS
SELECT empno,ename,sal FROM emp WHERE sal>2500;

--create a view on empno,sal,comm and total income
CREATE VIEW v_totalincome AS
SELECT empno,ename,sal,comm,sal+IFNULL(comm,0) AS total_income FROM emp;
```

```
--create a view on job,count(emps),sum(sal),max,min,avg of sal
CREATE VIEW v_jobsummary AS
SELECT job,COUNT(empno) empcount, SUM(sal) salsum,MAX(sal) maxsal,MIN(sal) minsal,
AVG(sal) avgsal FROM emp GROUP BY job;

INSERT INTO emp(empno,ename,sal,deptno) VALUES(1000,'JILL',2000,40);
```

```
SELECT * FROM v_richemp;

INSERT INTO v_richemp(empno,ename,sal) VALUES(1001,'JAMES',3500);
SELECT * FROM v_richemp; -- james will be visible
SELECT * FROM emp;

INSERT INTO v_richemp(empno,ename,sal) VALUES(1002,'ROCK',2200);
SELECT * FROM v_richemp; -- ROCK will not be visible
SELECT * FROM emp;

ALTER VIEW v_richemp AS
SELECT empno,ename,sal FROM emp WHERE sal>2500
WITH CHECK OPTION;

INSERT INTO v_richemp(empno,ename,sal) VALUES(1003,'HARRY',2100);
-- cannot insert as check option will fail

--can we insert the data into v_jobsummary?
SELECT * FROM v_jobsummary;
```

## Creating view from another view

```
CREATE VIEW v_richemp2 AS
SELECT empno,ename,sal FROM v_richemp;

SHOW TABLES;
SHOW CREATE VIEW v_richemp;
SHOW CREATE VIEW v_richemp2;

INSERT INTO v_richemp2(empno,ename,sal) VALUES(1004,'BRUCE',2300);
--error

DROP VIEW v_richemp;

SELECT * FROM v_richemp2;--error
--reference table/view invalid

DROP VIEW v_richemp2;
```

## Creating view using join

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

--create a view for above requirement
CREATE VIEW v_empdept AS
SELECT e.ename,e.sal,d.deptno,d.dname FROM emp e
INNER JOIN dept d ON e.deptno=d.deptno;

--diplay all emps from 'accounting' dept
SELECT e.ename,e.sal,d.deptno,d.dname FROM emp e
INNER JOIN dept d ON e.deptno=d.deptno WHERE d.dname='ACCOUNTING';

SELECT * FROM v_empdept WHERE dname="ACCOUNTING";
```

# INDEXES

1. SIMPLE INDEX

```
SELECT * FROM books;

--i want to display all books of C Programming
SELECT * FROM books WHERE subject='C Programming';

EXPLAIN FORMAT=JSON SELECT * FROM books WHERE subject='C Programming';
--1.45

DESC books;

CREATE INDEX idx_books_subject ON books(subject);

DESC books;

SELECT * FROM books WHERE subject='C Programming';
EXPLAIN FORMAT=JSON SELECT * FROM books WHERE subject='C Programming';
--0.90

SELECT * FROM books WHERE author = 'Yashwant Kanetkar';

EXPLAIN FORMAT=JSON SELECT * FROM books WHERE author = 'Yashwant Kanetkar';

CREATE INDEX idx_books_author ON books(author DESC);
DESC books;

--display ename,sal,deptno,dname.
EXPLAIN FORMAT = JSON SELECT e.ename,e.sal,d.deptno,d.dname FROM emp e
INNER JOIN dept d ON e.deptno=d.deptno;
--7.70

CREATE INDEX idx_emp_deptno ON emp(deptno);
```

```
CREATE INDEX idx_dept_deptno ON dept(deptno);

EXPLAIN FORMAT = JSON SELECT e.ename,e.sal,d.deptno,d.dname FROM emp e
INNER JOIN dept d ON e.deptno=d.deptno;
--4.01
```

2. UNIQUE INDEX
    - Duplicate values are not allowed in that column
    - Multiple NULL values are allowed

```
DESC emp;
CREATE UNIQUE INDEX idx_emp_ename ON emp(ename);
DESC emp;

INSERT INTO emp(empno,ename) VALUES(1000,'JAMES');
--error

INSERT INTO emp(empno,sal) VALUES(1000,1000);
INSERT INTO emp(empno,sal) VALUES(1001,2000);

CREATE UNIQUE INDEX idx_emp_mgr ON emp(mgr);
--error cannot create unique index on columns which have duplicate values.

SHOW INDEXES FROM emp;
```

3. CLUSTERED INDEX
    - It is a index that is automatically created on primary key
    - It is a unique index
    - If primary key does not exists in table then clusterd index is created on hidden(synthetic) coloumn

```
SHOW INDEXES FROM emp;
DROP INDEX idx_emp_ename ON emp;

SHOW INDEXES FROM books;
DROP INDEX idx_books_author ON books;
```

# Index

- Index enable faster searching in tables by indexed columns.
  - CREATE INDEX idx_name ON table(column);
- One table can have multiple indexes on different columns/order.
- Typically indexes are stored as some data structure (like BTREE or HASH) on disk.
- Indexes are updated during DML operations. So DML operation are slower on indexed tables.

# Index

- Index can be ASC or DESC.
  - It cause storage of key values in respective order (MySQL 8.x onwards).
  - ASC/DESC index is used by optimizer on ORDER BY queries.
- There are four types of indexes:
  - Simple index
    - CREATE INDEX idx_name ON table(column [ASC|DESC]);
  - Unique index
    - CREATE UNIQUE INDEX idx_name ON table(column [ASC|DESC]);
    - Doesn't allow duplicate values.
  - Composite index
    - CREATE INDEX idx_name ON table(column1 [ASC|DESC], column2 [ASC|DESC]);
    - Composite index can also be unique. Do not allow duplicate combination of columns.
  - Clustered index
    - PRIMARY index automatically created on Primary key for row lookup.
    - If primary key is not available, hidden index is created on synthetic column.
    - It is maintained in tabular form and its reference is used in other indexes.

# Index

- Indexes should be created on shorter (INT, CHAR, …) columns to save disk space.
- Few RDBMS do not allow indexes on external columns i.e. TEXT, BLOB.
- MySQL support indexing on TEXT/BLOB up to n characters.
  - CREATE TABLE test (blob_col BLOB, …, INDEX(blob_col(10)));
- To list all indexes on table:
  - SHOW INDEXES FROM table;
- To drop an index:
  - DROP INDEX idx_name ON table;
- When table is dropped, all indexes are automatically dropped.
- Indexes should not be created on the columns not used frequent search, ordering or grouping operations.
- Columns in join operation should be indexed for better performance.

# Constraints

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

# Constraints

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

# Constraints

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

# Constraints

- FOREIGN KEY
  - Column or set of columns that references a column of some table.
  - If column belongs to the same table, it is "self referencing".
  - Foreign key constraint is specified on child table column.
  - FK can have duplicate values as well as null values.
  - FK constraint is applied on column of child table (not on parent table).
  - Child rows cannot be deleted, until parent rows are deleted.
  - MySQL have ON DELETE CASCADE clause to ensure that child rows are automatically deleted, when parent row is deleted. ON UPDATE CASCADE clause does same for UPDATE operation.
  - By default foreign key checks are enabled. They can be disabled by
    - SET @@foreign_key_checks = 0;
  - FK constraint can be applied on table level as well as column level.
  - CREATE TABLE child(c1 TYPE, …, FOREIGN KEY (c1) REFERENCES parent(col))

# Constraints

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

# DDL – ALTER statement

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

# Agenda

```
-Indexes
    - Composite Index
- Constraints
    - NOT NULL
    - UNIQUE
    - PRIMARY KEY
    - FOREIGN KEY
    - CHECK
- ALTER
```

4. Composite Index
    - It is a index that is applied on two columns

```
--display emp from dept 20;
SELECT * FROM emp Where deptno=20;

--display emp from job CLERK;
SELECT * FROM emp Where job='CLERK';

--display all emps who are working as clerk in dept 20
SELECT * FROM emp Where deptno=20 AND JOB='CLERK';

EXPLAIN FORMAT=JSON SELECT * FROM emp Where deptno=20 AND JOB='CLERK';
--1.45

CREATE INDEX idx_emp_dj ON emp(deptno ASC,job ASC);

SHOW INDEXES FROM emp

EXPLAIN FORMAT=JSON SELECT * FROM emp Where deptno=20 AND JOB='CLERK';
--0.70

--check query cost for finding emp with sal 5000;
EXPLAIN FORMAT=JSON SELECT * FROM emp WHERE sal=5000;
--1.45

--check query cost for finding emp with dept 20;
EXPLAIN FORMAT=JSON SELECT * FROM emp WHERE deptno=20;
--1.00

--check query cost for finding emp with job clerk;
EXPLAIN FORMAT=JSON SELECT * FROM emp WHERE job='clerk';
--1.45

CREATE INDEX idx_emp_job ON emp(job ASC);

EXPLAIN FORMAT=JSON SELECT * FROM emp WHERE job='clerk';
```

```
--0.90

CREATE TABLE students(std INT, roll INT, name CHAR(30), marks DECIMAL(5,2));
INSERT INTO students VALUES (1, 1, 'Soham', 99);
INSERT INTO students VALUES (1, 2, 'Sakshi', 96);
INSERT INTO students VALUES (1, 3, 'Prisha', 98);
INSERT INTO students VALUES (2, 1, 'Madhu', 97);
INSERT INTO students VALUES (2, 2, 'Om', 95);

CREATE UNIQUE INDEX idx_std_roll ON students(std ASC,roll ASC);
INSERT INTO students VALUES (2, 2, 'Rahul', 95);--error duplicate values
INSERT INTO students VALUES (2, 1, 'Rahul', 95);--error duplicate values
INSERT INTO students VALUES (1, 3, 'Rahul', 95);--error duplicate values

INSERT INTO students VALUES (2, 3, 'Rahul', 95);
```

## Constraints

```
- Constraints are checked/verified when you do any DML operations.
- It will slow down the DML operations
- It helps to enter the valid and correct data into the tables.
- You can put the constraints on column level or on table level.
- except NOT NULL all the other constraints can be applied on column level as well
as on table level.
```

1. NOT NULL
    - If you want to check for any column that should not have null values while performing DML we use use NOT NULL Constraint

```
CREATE TABLE temp(c1 INT,c2 INT, c3 INT NOT NULL);
INSERT INTO temp VALUES(1,1,1);
INSERT INTO temp(c2,c3) VALUES(2,2);
INSERT INTO temp(c1,c3) VALUES(3,3);
INSERT INTO temp(c1,c2) VALUES(4,4); --error
```

2. UNIQUE
    - If you want to keep unique vaules in column then we should use unique constraint
    - Unique can have multiple NULL values but repetation of values are not allowed.
    - Unique constraint on combination of multiple columns internally creates Composite Unique index.
    - Must be at table level -- UNIQUE(c1,c2).

```
CREATE TABLE temp1(c1 INT UNIQUE,c2 INT, c3 INT);

DESC temp1;
```

```
SHOW INDEXES FROM temp1;

INSERT INTO temp1 VALUES(1,1,1);
INSERT INTO temp1 VALUES(2,1,1);
SELECT * FROM temp1;
INSERT INTO temp1 VALUES(2,2,2);--error
INSERT INTO temp1(c2,c3) VALUES(3,3);
INSERT INTO temp1(c2,c3) VALUES(4,4);

DROP TABLE students;
CREATE TABLE students(std INT, roll INT, name CHAR(30), marks
DECIMAL(5,2),UNIQUE(std,roll));
--using Unique on table level

INSERT INTO students VALUES (1, 1, 'Soham', 99);
INSERT INTO students VALUES (1, 2, 'Sakshi', 96);
INSERT INTO students VALUES (1, 3, 'Prisha', 98);
INSERT INTO students VALUES (2, 1, 'Madhu', 97);
INSERT INTO students VALUES (2, 2, 'Om', 95);

DESC students;
INSERT INTO students VALUES (2, 2, 'Rahul', 95);--error duplicate values
INSERT INTO students VALUES (2, 1, 'Rahul', 95);--error duplicate values
INSERT INTO students VALUES (1, 3, 'Rahul', 95);--error duplicate values

INSERT INTO students(name,marks) VALUES ('Rahul', 95);
INSERT INTO students(std,name,marks) VALUES (1,'Pratik', 85);
INSERT INTO students(std,name,marks) VALUES (1,'Onkar', 75);
INSERT INTO students VALUES (2, 3, 'Vishaka', 95);
SELECT * FROM students;
```

3. Primary Key
    - Unique + NOT NULL
    - In a table you can a single primary key but you can have multiple unique constraints.

```
CREATE TABLE cdac_students(
prnno INT PRIMARY KEY,
name CHAR(30),
email VARCHAR(50),
mobileno CHAR(10),
marks INT
);

CREATE TABLE temp2(c1 INT PRIMARY KEY,c2 INT,c3 INT);
DESC temp2;
SHOW INDEXES FROM temp2;

INSERT INTO temp2 VALUES(1,1,1);
INSERT INTO temp2 VALUES(2,1,1);
INSERT INTO temp2 VALUES(2,3,3); -- error
INSERT INTO temp2 VALUES(NULL,3,3); -- error
```

## Composite Primary Key

The primary key can be combination multiple columns. It is called as Composite Primary Key.

```
DROP TABLE students;

CREATE TABLE students(std INT, roll INT, name CHAR(30), marks DECIMAL(5,2),PRIMARY
KEY(std,roll));

DESC students;

SHOW INDEXES FROM students;

INSERT INTO students VALUES (1, 1, 'Soham', 99);
INSERT INTO students VALUES (1, 2, 'Sakshi', 96);
INSERT INTO students VALUES (1, 3, 'Prisha', 98);
INSERT INTO students VALUES (2, 1, 'Madhu', 97);
INSERT INTO students VALUES (2, 2, 'Om', 95);

INSERT INTO students(name,marks) VALUES ('Rahul', 95);--error
INSERT INTO students(std,name,marks) VALUES (1,'Pratik', 85);--error
INSERT INTO students(std,roll,name,marks) VALUES (1,2,'Onkar', 75);--error
INSERT INTO students VALUES (2, 3, 'Vishaka', 95);
```

## Surrogate Primary Key

```
Usually auto-generated.
ORACLE->Sequences
MSSQL -> Identity
MySQL -> Auto_Increment
```

```
CREATE TABLE products(id INT PRIMARY KEY AUTO_INCREMENT, name CHAR(20),price
DECIMAL(9,2));

INSERT INTO products(name,price) VALUES('sugar',35);
INSERT INTO products(name,price) VALUES('Lemon',10);
INSERT INTO products(name,price) VALUES('Tata Salt',25);
INSERT INTO products(name,price) VALUES('Almonds',540);

SELECT * FROM products;

ALTER TABLE PRODUCTS AUTO_INCREMENT=100;

INSERT INTO products(name,price) VALUES('Maggi',15);
INSERT INTO products(name,price) VALUES('Amul Cheese',115);
```

4. Foreign key

```
DROP TABLE emps;
DROP TABLE depts;

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

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

INSERT INTO emps VALUES (1, 'Amit', 10, 4);
INSERT INTO emps VALUES (2, 'Rahul', 10, 3);
INSERT INTO emps VALUES (3, 'Nilesh', 20, 4);

INSERT INTO emps VALUES (4, 'Nitin', 50, 5);
--error foreign key constraint fails

INSERT INTO emps VALUES (5, 'Sarang', 50, NULL);
--error foreign key constraint fails

INSERT INTO emps VALUES (4, 'Nitin', 30, 5);
INSERT INTO emps VALUES (5, 'Sarang', 30, NULL);
INSERT INTO emps VALUES (6, 'Rohan', NULL, NULL);


SELECT * FROM emps;
SELECT * FROM depts;

DELETE FROM depts WHERE deptno=40;

DELETE FROM depts WHERE deptno=30;
--error foreign key constarint fails

DROP TABLE depts;
--error foreign key constraint fails

DROP TABLE emps;
DROP TABLE depts;
```

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

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

INSERT INTO emps VALUES (1, 'Amit', 10, 4);
INSERT INTO emps VALUES (2, 'Rahul', 10, 3);
INSERT INTO emps VALUES (3, 'Nilesh', 20, 4);
INSERT INTO emps VALUES (4, 'Nitin', 30, 5);
INSERT INTO emps VALUES (5, 'Sarang', 30, NULL);

UPDATE depts SET deptno=50 WHERE deptno=30;
--changes will happen in depts as well as in emps

DELETE FROM depts WHERE deptno=50;
--changes will happen in depts as well as in emps


CREATE TABLE stu(std INT, roll INT, name CHAR(30),PRIMARY KEY(std,roll));

INSERT INTO stu VALUES (1, 1, 'Soham');
INSERT INTO stu VALUES (1, 2, 'Sakshi');
INSERT INTO stu VALUES (1, 3, 'Prisha');
INSERT INTO stu VALUES (2, 1, 'Madhu');
INSERT INTO stu VALUES (2, 2, 'Om');

CREATE TABLE marks(std INT, roll INT, marks DECIMAL(5,2),FOREIGN KEY (std,roll)
REFERENCES stu(std,roll));

INSERT INTO marks VALUES (1, 1, 50);
INSERT INTO marks VALUES (1, 2, 60);
INSERT INTO marks VALUES (1, 3, 70);
INSERT INTO marks VALUES (2, 1, 80);
INSERT INTO marks VALUES (2, 2, 90);

CREATE TABLE depts (
deptno INT,
dname VARCHAR(20),
PRIMARY KEY(deptno)
```

```
);
INSERT INTO depts VALUES (10, 'DEV');
INSERT INTO depts VALUES (20, 'QA');
INSERT INTO depts VALUES (30, 'OPS');
INSERT INTO depts VALUES (40, 'ACC');

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


INSERT INTO emps VALUES (1, 'Amit', 10, 4);
INSERT INTO emps VALUES (2, 'Rahul', 10, 3);
INSERT INTO emps VALUES (3, 'Nilesh', 20, 4);

INSERT INTO emps VALUES (4, 'Nitin', 50, 5);
--cannot insert

SELECT @@foreign_key_checks; --1

SET @@foreign_key_checks=0;

SELECT @@foreign_key_checks; --0

--Now you can insert the data of dept 50 as the foreign key checks are disabled.
INSERT INTO emps VALUES (4, 'Nitin', 50, 5);
INSERT INTO emps VALUES (5, 'Sarang', 50, NULL);

SET @@foreign_key_checks=1;
INSERT INTO emps VALUES (6, 'Rohan', 60, 4);

CREATE TABLE depts_backup (
deptno INT,
dname VARCHAR(20),
PRIMARY KEY(deptno)
);

CREATE TABLE emps_backup(
empno INT PRIMARY KEY,
ename VARCHAR(20),
deptno INT,
mgr INT,
FOREIGN KEY(deptno) REFERENCES depts_backup(deptno)
);

SET @@foreign_key_checks=0;

INSERT INTO emps_backup SELECT empno,ename,deptno,mgr FROM emp;

SET @@foreign_key_checks=1;
```

```
--self referencing Foreign Key
CREATE TABLE emps2(
empno INT PRIMARY KEY,
ename VARCHAR(20),
deptno INT,
mgr INT,
FOREIGN KEY(deptno) REFERENCES depts(deptno),
FOREIGN KEY(mgr) REFERENCES emps2(empno)
);
```

5. CHECK

```
CREATE TABLE voters(
name CHAR(20),
age INT NOT NULL CHECK (age>18)
);

INSERT INTO voters VALUES('Rohan',28);--OK
INSERT INTO voters VALUES('Ranbir',16);--NOT OK
INSERT INTO voters VALUES('Alia',NULL);--NOT OK
```

## Contraint names

```
SHOW CREATE TABLE emps;

--ColOUMN LEVEL CONSTRAINT
CREATE TABLE customer(
adharcard CHAR(12) PRIMARY KEY,
name CHAR(30) NOT NULL,
email VARCHAR(50) UNIQUE,
password CHAR(20) NOT NULL CHECK(LENGTH(password)>8),
mobile CHAR(10) UNIQUE NOT NULL,
age INT NOT NULL
);

CREATE TABLE ADDRESS(
id INT PRIMARY KEY AUTO_INCREMENT,
adharcard CHAR(12),
buildingno CHAR(5),
city CHAR(15) NOT NULL,
pin INT NOT NULL,
landmark VARCHAR(20),
FOREIGN KEY adharcard REFERENCES customer(adharcard)
);


CREATE TABLE customer(
```

```
adharcard CHAR(12),
name CHAR(30) NOT NULL,
email VARCHAR(50),
password CHAR(20) NOT NULL,
mobile CHAR(10) NOT NULL,
age INT NOT NULL,
CONSTRAINT pk_adharcard PRIMARY KEY(adharcard),
CONSTRAINT un_email UNIQUE(email),
CONSTRAINT ch_password CHECK(LENGTH(password)>8),
CONSTRAINT un_mobile UNIQUE(mobile)
);

CREATE TABLE ADDRESS(
id INT PRIMARY KEY AUTO_INCREMENT,
adharcard CHAR(12),
buildingno CHAR(5),
city CHAR(15) NOT NULL,
pin INT NOT NULL,
landmark VARCHAR(20),
CONSTRAINT fk_adharcard FOREIGN KEY adharcard REFERENCES customer(adharcard)
);
```

## ALTER

```
- Alter is used to modify the structure of table
- Not recommended when database is in production level
- After alter you table structure becomes inefficient
```

```
DESC emps;

--add a column job in the emps table
ALTER TABLE emps ADD COLUMN job VARCHAR(20);
DESC emps;
SELECT * FROM emps;

--change the datatype of job
ALTER TABLE emps MODIFY job VARCHAR(30);
DESC emps;

--change name of column
ALTER TABLE emps CHANGE ename name varchar(20);
DESC emps;

--Remove a column job
ALTER TABLE emps DROP COLUMN job;
DESC emps;

--Add a constraint
ALTER TABLE emps ADD CONSTRAINT UNIQUE(name);
```

```
DESC emps;

--DELETE a constraint
ALTER TABLE emps DROP CONSTRAINT name;
ALTER TABLE emps DROP CONSTRAINT emps_ibfk_1;
DESC emps;

--rename a table
```

# MySQL Programming

- RDBMS Programming is an ISO standard – part of SQL standard – since 1992.
- SQL/PSM stands for Persistent Stored Module.
- Inspired from PL/SQL - Programming language of Oracle.
- PSM allows writing programs for RDBMS. The program contains set of SQL statements along with programming constructs e.g. variables, if-else, loops, case, ...
- PSM is a block language. Blocks can be nested into another block.
- MySQL program can be a stored procedure, function or trigger.

# MySQL Programming

- MySQL PSM program is written by db user (programmers).
- It is submitted from client, server check syntax & store them into db in compiled form.
- The program can be executed by db user when needed.
- Since programs are stored on server in compiled form, their execution is very fast.
- All these programs will run in server memory.

# Stored Procedure

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

# Stored Procedure

```
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;
```

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


SOURCE /path/to/01_hello.sql

CALL sp_hello();
```

# Stored Procedure – 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(PARAMTYPE p1 DATATYPE)
BEGIN
    …
END;

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

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

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

## IF-ELSE

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

## LOOPS

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

## CASE-WHEN

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

## SHOW PROCEDURE

```
SHOW PROCEDURE STATUS
LIKE ‘sp_name’;

SHOW CREATE PROCEDURE sp_name;
```

## DROP PROCEDURE

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

# MySQL Exceptions

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

# MySQL Stored Functions

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

**CREATE FUNCTION**

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

**SHOW FUNCTION**

```
SHOW FUNCTION STATUS LIKE 'fn_name';

SHOW CREATE FUNCTION fn_name;
```

**DROP FUNCTION**

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

# AGENDA

```
PL/SQL
Exception Handling
Cursor
Functions
```

# PL/SQL

```
Procedural Language
PSM - Persistant Stored Modules
```

# Delimiter

```
Default delimiter is  ; in mysql
When ; is found the code is submitted to mysql server and it is processed.
It needs to be changed temporarily for the Stored Procedure using DELIMITER
keyword.
```

```
DELIMITER $$
SELECT * FROM emp; -- Wil not execute
$$ -- the statement will now be executed

DELIMITER ; -- Changing back the delimiter
```

# Steps for stored Procedure Programming

```
1. Create a file with extension as .sql
2. Write the procedure follwing the proper syntax
3. Save the File
4. use SOURCE cmd to import the file in mysql
5. CALL the procedure
```

# To look for all procedures use below query

```
 SHOW PROCEDURE STATUS WHERE db='classwork';
```

```
CREATE TABLE result(id INT, msg CHAR(20));
```

## Errors in Mysql

```
1. Error code
    1045 - Access denied
    1062 - Duplicate entry
    1146 - Table doesn't exist

2. Error State
    28000 - Access denied
    23000 - Duplicate entry
    42S02 - Table doesn't exist
```

## Exception Handling

```
CREATE PROCEDURE sp_placeorder()
BEGIN

DECLARE EXIT HANDLER FOR error
BEGIN
ROLLBACK;
SELECT 'Order Failed - Try Again' AS msg;
END;

START TRANSACTION;
INSERT INTO orders VALUES(...);
INSERT INTO orders_items VALUES(...);
INSERT INTO payment VALUES(...);
COMMIT;

END;
```

## CURSOR

Cursor is a special variable in PSM used to access rows/values "one by one" from result of "SELECT" statement.

## Programming Steps

1. Declare handler for end of cursor (like end-of-file). Error code: "NOT FOUND".
2. Declare cursor variable with its SELECT statement.
3. Open cursor.
4. Fetch (current row) values from cursor into some variables & process them.
5. Repeat process all rows in SELECT output. At the end error handler will be executed.
6. Exit the loop and close the cursor.

# Characteristics of "MySQL Cursors"

1. Readonly
    - We can use cursor only for reading from the table.
    - Cannot update or delete from the cursor.
    - SET v_cur1 = (1, 'NewName'); -- not allowed
    - To update or delete programmer can use UPDATE/DELETE queries.
2. Non-scrollable
    - Cursor is forward only.
    - Reverse traversal or random access of rows is not supported.
    - When FETCH is done, current row is accessed and cursor automatically go to next row.
    - We can close cursor and reopen it. Now it again start iterating from the start.
3. Asensitive
    - When cursor is opened, the addresses of all rows (as per SELECT query) are recorded into the cursor (internally). These rows are accessed one by one (using FETCH).
    - While cursor is in use, if other client modify any of the rows, then cursor get modified values.
    - Because cursor is only having address of rows, Cursor is not creating copy of the rows. Hence MySQL cursors are faster.

# Functions

- User-defined Functions

1. DETERMINISTIC
    - If input is same, output will remain same ALWAYS.
    - Internally MySQL cache input values and corresponding output.
    - If same input is given again, directly output may return to speedup execution.
2. NOT DETERMINISTIC
    - Even if input is same, output may differ.
    - Output also depend on current date-time or state of table or database settings.
    - These functions cannot be speedup.
    - Before importing NOT Deterministic Function do the below Setting.
    - SET GLOBAL @@log_bin_trust_function_creators=1;

# MySQL Triggers

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

**CREATE TRIGGER**

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

**SHOW TRIGGERS**

```
SHOW TRIGGERS FROM db_name;
```

**DROP TRIGGER**

```
DROP TRIGGER trig_name;
```

# Codd's rules

- Proposed by Edgar F. Codd – pioneer of the RDBMS – in 1980.
- If any DBMS follow these rules, it can be considered as RDBMS.

- The $0^{th}$ rule is the main rule known as "The foundation rule".
  - For any system that is advertised as, or claimed to be, a relational data base management system, that system must be able to manage data bases entirely through its relational capabilities.

- The rest of rules can be considered as elaboration of this foundation rule.

# Codd's rules

- Rule 1: The information rule:
  - All information in a relational data base is represented explicitly at the logical level and in exactly one way – by values in tables.
- Rule 2: The guaranteed access rule:
  - Each and every datum (atomic value) in a relational data base is guaranteed to be logically accessible by resorting to a combination of table name, primary key value and column name.
- Rule 3: Systematic treatment of null values:
  - Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type.

# Codd’s rules

- Rule 4: Dynamic online catalog based on the relational model:
  - The data base description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.

- Rule 5: The comprehensive data sublanguage rule:
  - A relational system may support several languages. However, there must be at least one language that supports all functionalities of a RDBMS i.e. data definition, data manipulation, integrity constraints, transaction management, authorization.

# Codd's rules

- Rule 6: The view updating rule:
  - All views that are theoretically updatable are also updatable by the system.
- Rule 7: Possible for high-level insert, update, and delete:
  - The capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update and deletion of data. Also it Should supports Union, Intersection Operation
- Rule 8: Physical data independence:
  - Application programs and terminal activities remain logically unbroken whenever any changes are made in either storage representations or access methods.
- Rule 9: Logical data independence:
  - Application programs & terminal activities remain logically unbroken when information-preserving changes of any kind that theoretically permit un-impairment are made to the base tables.

# Codd's rules

- Rule 10: Integrity independence:
  - Integrity constraints specific to a particular relational database must be definable in the relational data sublanguage and storable in the catalog, not in the application programs.
- Rule 11: Distribution independence:
  - The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only.
- Rule 12: The non-subversion rule:
  - If a relational system has a low-level (single-record-at-a-time) language, that low level cannot be used to subvert or bypass the integrity rules and constraints expressed in the higher level relational language (multiple-records-at-a-time).

# Normalization

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

# 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 desc of fields properties.
- Remove computed columns.
- Assign primary key to the table.
- At this stage data is in un-normalized form.
- UNF is starting point of normalization.

# Normalization

- UNF suffers from
  - Insert anomaly
  - Update anomaly
  - Delete anomaly

# Normalization

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

# Normalization

- 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.

# Normalization

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

# Normalization

- 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(Boyce-Codd Normal Form) or 4-NF.

# De-normalization

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

# Agenda

```
Triggers
Codd's Rule
Normalization
Introduction to Mongo
```

# Triggers

```
Triggers in MySQL are Programs(PSM Syntax).
It is executed(triggered) by some event.
DML Operations
AFTER INSERT
AFTER UPDATE
AFTER DELETE
BEFORE INSERT
BEFORE UPDATE
BEFORE DELETE

IF DML operation is done on multiple rows then for each individual row a trigger
will be fired.

NEW and OLD are the keywords.
These keywords are used on affected rows.
INSERT -> NEW
DELETE -> OLD
UPDATE -> NEW ,OLD

Triggers cannot be called explicitly by user.
```

```sql
DROP TABLE accounts;
CREATE TABLE accounts(
id INT,
type CHAR(15),
balance DECIMAL(10,2)
);

INSERT INTO accounts VALUES(1,'SAVINGS',0);
INSERT INTO accounts VALUES(2,'SAVINGS',0);
INSERT INTO accounts VALUES(3,'CURRENT',0);

CREATE TABLE transactions(
    tid INT PRIMARY KEY AUTO_INCREMENT,
    type CHAR(15),
    acc_id INT,
```

```
    amount DECIMAL(10,2)
);

INSERT INTO transactions(type,acc_id,amount) VALUES('Deposit',1,5000);
INSERT INTO transactions(type,acc_id,amount) VALUES('Deposit',2,8000);
INSERT INTO transactions(type,acc_id,amount) VALUES('Deposit',3,10000);

INSERT INTO transactions(type,acc_id,amount) VALUES('Withdraw',2,2000);
```

# Stored Procedure – 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(PARAMTYPE p1 DATATYPE)
BEGIN
    …
END;

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

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

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

## IF-ELSE

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

## LOOPS

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

## CASE-WHEN

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

## SHOW PROCEDURE

```
SHOW PROCEDURE STATUS
LIKE ‘sp_name’;

SHOW CREATE PROCEDURE sp_name;
```

## DROP PROCEDURE

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

# Agenda

- Introduction
- RDBMS vs NoSQL
- Scaling
- CAP theorem
- Advantages
- Limitations
- Applications
- Key-value database
- Column-oriented database
- Graph database
- Document database

# Database

- A database is an organized collection of data, generally stored and accessed electronically from a computer system
- A database refers to a set of related data and the way it is organized
- Database Management System
  - Software that allows users to interact with one or more databases and provides access to all of the data contained in the database
- Types
  - RDBMS
  - NoSQL

# RDBMS

- The idea of RDBMS was borne in 1970 by E. F. Codd.
- Structured and organized data
- Structured query language (SQL)
- DML, DQL, DDL, DTL, DCL.
- Data and its relationships are stored in separate tables.
- Tight Consistency
- Based on Codd’s rules
- ACID transactions.
  - Atomic
  - Consistent
  - Isolated
  - Durable

# NoSQL

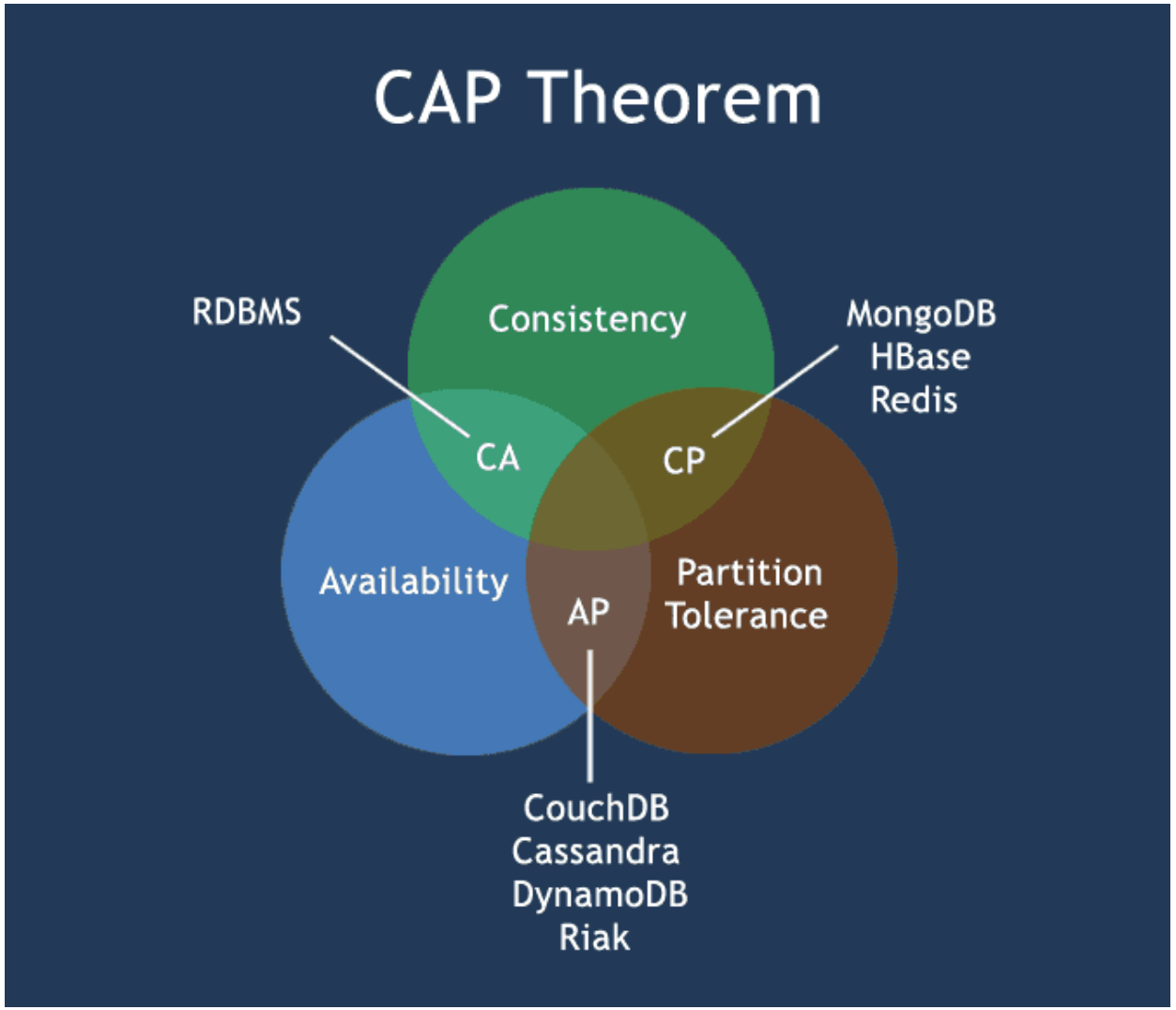
- Refer to non-relational databases
- Stands for Not Only SQL
- Term NoSQL was first used by Carlo Strozzi in 1998.
- No declarative query language
- No predefined schema, Unstructured and unpredictable data
- Eventual consistency rather ACID property
- Based on CAP Theorem
- Prioritizes high performance, high availability and scalability
- BASE Transaction
  - Basically Available
  - Soft state
  - Eventual consistency

# Scaling

- Scalability is the ability of a system to expand to meet your business needs.
- E.g. scaling a web app is to allow more people to use your application.
- Types of scaling
  - Vertical scaling: Add resources within the same logical unit to increase capacity. E.g. add CPUs to an existing server, increase memory in the system or expanding storage by adding hard drives.
  - Horizontal scaling: Add more nodes to a system. E.g. adding a new computer to a distributed software application. Based on principle of distributed computing.
- NoSQL databases are designed for Horizontal scaling. So they are reliable, fault tolerant, better performance (at lower cost), speed.

# CAP (Brewer's) Theorem

- **Consistency** - Data is consistent after operation. After an update operation, all clients see the same data.
- **Availability** - System is always on (i.e. service guarantee), no downtime.
- **Partition Tolerance** - System continues to function even the communication among the servers is unreliable.
- Brewer's Theorem
  - It is impossible for a distributed data store to simultaneously provide more than two out of the above three guarantees.



# Advantages of NoSQL

- High scalability
  - This scaling up approach fails when the transaction rates and fast response requirements increase. In contrast to this, the new generation of NoSQL databases is designed to scale out (i.e. to expand horizontally using low-end commodity servers).
- Manageability and administration
  - NoSQL databases are designed to mostly work with automated repairs, distributed data, and simpler data models, leading to low manageability and administration.
- Low cost
  - NoSQL databases are typically designed to work with a cluster of cheap commodity servers, enabling the users to store and process more data at a low cost.
- Flexible data models
  - NoSQL databases have a very flexible data model, enabling them to work with any type of data; they don't comply with the rigid RDBMS data models. As a result, any application changes that involve updating the database schema can be easily implemented.

# Disadvantages of NoSQL

- Maturity
  - Most NoSQL databases are pre-production versions with key features that are still to be implemented. Thus, when deciding on a NoSQL database, you should analyse the product properly to ensure the features are fully implemented and not still on the To-do list.
- Support
  - Support is one limitation that you need to consider. Most NoSQL databases are from start-ups which were open sourced. As a result, support is very minimal as compared to the enterprise software companies and may not have global reach or support resources.
- Limited Query Capabilities
  - Since NoSQL databases are generally developed to meet the scaling requirement of the web-scale applications, they provide limited querying capabilities. A simple querying requirement may involve significant programming expertise.
- Administration
  - Although NoSQL is designed to provide a no-admin solution, it still requires skill and effort for installing and maintaining the solution.
- Expertise
  - Since NoSQL is an evolving area, expertise on the technology is limited in the developer and administrator community.

# Applications

- When to use NoSQL?
  - Large amount of data (TBs)
  - Many Read/Write ops
  - Economical Scaling
  - Flexible schema
- Examples:
  - Social media
  - Recordings
  - Geospatial analysis
  - Information processing

- When Not to use NoSQL?
  - Need ACID transactions
  - Fixed multiple relations
  - Need joins
  - Need high consistency
- Examples
  - Financial transactions
  - Business operations

# RDBMS vs NoSQL

| | RDBMS | NoSQL |
|---|---|---|
| **Types** | All types support SQL standard | Multiple types exists, such as document stores, key value stores, column databases, etc |
| **History** | Developed in 1970 | Developed in 2000s |
| **Examples** | SQL Server, Oracle, MySQL | MongoDB, HBase, Cassandra, Redis, Neo4J |
| **Data Storage Model** | Data is stored in rows and columns in a table, where each column is of a specific type | The data model depends on the database type. It could be Key-value pairs, documents etc |
| **Schemas** | Fixed structure and schema | Dynamic schema. Structures can be accommodated |
| **Scalability** | Scale up approach is used | Scale out approach is used |
| **Transactions** | Supports ACID and transactions | Supports partitioning and availability |
| **Consistency** | Strong consistency | Dependent on the product [Eventual Consistency] |
| **Support** | High level of enterprise support | Open source model |
| **Maturity** | Have been around for a long time | Some of them are mature; others are evolving |

# NoSQL database

- NoSQL databases are non-relational.
- There is no standardization/rules of how NoSQL database to be designed.
- All available NoSQL databases can be broadly categorized as follows:
  - Key-value databases
  - Column-oriented databases
  - Graph databases
  - Document oriented databases

# Key-value database

- Based on Amazon's Dynamo database.
- For handling huge data of any type.
- Keys are unique and values can be of any type i.e. JSON, BLOB, etc.
- Implemented as big distributed hash-table for fast searching.
- Example: redis, dynamodb, riak, ...

# Column-oriented databases

- Values of columns are stored contiguously.
- Better performance while accessing few columns and aggregations.
- Good for data-warehousing, business intelligence, CRM, ...
- Examples: hbase, cassandra, bigtable, …

# Graph databases

- Graph is collection of vertices and edges (lines connecting vertices).
- Vertices keep data, while edges represent relationships.
- Each node knows its adjacent nodes. Very good performance, when want to access all relations of an entity (irrespective of size of data).
- Examples: Neo4J, Titan, …

# Document oriented databases

- Document contains data as key-value pair as JSON or XML.
- Document schema is flexible & are added in collection for processing.
- RDBMS tables → Collections
- RDBMS rows → Documents
- RDBMS columns → Key-value pairs in document
- Examples: MongoDb, CouchDb, …


# MongoDb Databases

Trainer: Mr. Rohan Paramane

# Mongo Db

- Developed by 10gen in 2007
- Publicly available in 2009
- Open-source database which is controlled by 10gen
- Document oriented database → stores JSON documents
- Stores data in binary JSON.
- Design Philosophy
  - MongoDB wasn’t designed in a lab and is instead built from the experiences of building large scale, high availability, and robust systems.

# Mongo Server and Client

- MongoDb server (mongod) is developed in C, C++ and JS.
- MongoDb data is accessed via multiple client tools
  - mongo : client shell (JS).
  - mongofiles : stores larger files in GridFS.
  - mongoimport / mongoexport : tools for data import / export.
  - mongodump / mongorestore : tools for backup / restore.
- MongoDb data can be accessed in application through client drivers available for all major programming languages e.g. Java, Python, Ruby, PHP, Perl, …
- Mongo shell is follows JS syntax and allow to execute JS scripts.

# BSON

- BSON simply stands for “Binary JSON”
- Binary structure encodes type and length information, which allows it to be parsed much more quickly
- It has been extended to add some optional non-JSON-native data types
- It allows for comparisons and calculations to happen directly on data
- MongoDB stores data in BSON format both internally, and over the network
- Anything you can represent in JSON can be natively stored in MongoDB

| | JSON | BSON |
|---|---|---|
| **Encoding** | UTF-8 String | Binary |
| **Data Support** | • String<br>• Boolean<br>• Number<br>• Array | • String<br>• Boolean<br>• Number<br>  • Integer<br>  • Float<br>  • Long<br>  • Decimal<br>• Array<br>• Date<br>• Raw Binary |
| | Human and Machine | Machine Only |

# MongoDb: Data Types

| data | bson | values |
|---|---|---|
| null | 10 | |
| boolean | 8 | true, false |
| number | 1 / 16 / 18 | 123, 456.78, NumberInt(“24”), NumberLong(“28”) |
| string | 2 | “....” |
| date | 9 | new Date(), ISODate(“yyyy-mm-ddThh:mm:ss”) |
| array | 4 | [ …, …, …, … ] |
| object | 3 | { … } |

# MongoDB Terminology

- Database
  - Like database/schema in RDBMS.
  - mongo> show databases;
  - mongo> use dbname;
- Collection
  - Like table in RDBMS.
  - No fixed structure or schema.
  - mongo> db.createCollection("colname");
- Document
  - Like row in RDBMS.
  - Inserted in JSON format.
  - Each record can have different fields.
- Field
  - Like column in RDBMS.
  - A name-value pair in a document.

# Mongo - INSERT

- show databases;
- use database;
- db.contacts.insert({name: "nilesh", mobile: "9527331338"});
- db.contacts.insertMany([
  {name: "nilesh", mobile: "9527331338"},
  {name: "nitin", mobile: "9881208115"}
  ]);
- Maximum document size is 16 MB.
- For each object unique id is generated by client (if _id not provided).
  - 12 byte unique id :: [counter(3) | pid(2) | machine(3) | timestamp(4)]

# Mongo – QUERY

- db.contacts.find(); → returns cursor on which following ops allowed:
  - hasNext(), next(), skip(n), limit(n),  count(), toArray(), forEach(fn), pretty()
- Shell restrict to fetch 20 records at once. Press "it" for more records.
- db.contacts.find( { name: "nilesh" } );
- db.contacts.find( { name: "nilesh" }, { _id:0, name:1 } );
- Relational operators: $eq, $ne, $gt, $lt, $gte, $lte, $in, $nin
- Logical operators: $and, $or, $nor, $not
- Element operators: $exists, $type
- Evaluation operators: $regex, $where, $mod
- Array operators: $size, $elemMatch, $all, $slice

# Mongo – DELETE

- db.contacts.remove(criteria);
- db.contacts.deleteOne(criteria);
- db.contacts.deleteMany(criteria);
- db.contacts.deleteMany({}); → delete all docs, but not collection
- db.contacts.drop(); → delete all docs & collection as well : efficient

# Mongo – UPDATE

- db.contacts.update(criteria, newObj);
- Update operators: $set, $inc, $dec, $push, $each, $slice, $pull
- In place updates are faster (e.g. $inc, $dec, …) than setting new object. If new object size mismatch with older object, data files are fragmented.
- Update operators: $addToSet
- example: db.contacts.update( { name: "peter" },
- { $push : { mobile: { $each : ["111", "222" ], $slice : -3 } } } );
- db.contacts.update( { name: "t" }, { $set : { "phone" : "123" } }, true );
  - If doc with given criteria is absent, new one is created before update.

# Mongo - Indexes

- db.books.find( { "subject" : "C" } ).explain(true);
- explain() → explains the query execution plan.
- Above query by default does full collection scan, hence slower.
- db.books.createIndex( { "subject" : 1 } );
- Searching on indexed columns reduces query execution time.
- Options can be provided (2nd arg): { unique : true }
  - Duplicate values are not allowed in that field.
- By default "_id" field is indexed in mongodb (unique index).
- db.books.getIndexes();
- db.books.dropIndex({ "subject" : 1 });

# Agenda

```
NoSQL
BASE Transaction
JSON
Mongo
```

# BASE Transactions

```
BA - Basically Available
S - Soft State
E - Eventual Consistency
```

# Key-Value database

```
Key                 Value
"emp/1001/name"     "JAMES"
"emp/1001/sal"      2000
"emp/1001/job"      "CLERK"
"emp/1002/name"     "JILL"
"emp/1002/sal"      3000
"emp/1002/job"      "ANALYST"
"emp/1002/image"    image
```

# Column Oriented Database

```
ENAME       JOB             SAL
JAMES       CLERK           1000
JILL        ANALYST         2000
HARRY       DEVELOPER       3000
ROCK        PRESIDENT       4000
```

# Documnet Oriented database

```
- JSON -> Java Script Object Notation
- keys in json are always stored as string
- data inside JSON is always stored in key-value pair
- in mongo the below JSON syntax is called as Document
```

```
//JSON Syntax
{
    "key1":1
    "key2":"value2"
}

{
    "id" : 1,
    "name":"rohan",
    "sal":2000
}

{
    "id" : 2,
    "name":"pratik",
    "sal":3000,
    "comm": 200
}


{
    "id" : 3,
    "name":"Sunbeam",
    "website" : "www.sunbeaminfo.in"
    "emp1":{
    "id" : 2,
    "name":"pratik",
    "sal":3000,
    "comm": 200
    }
}

{
    "id" : 3,
    "name":"Sunbeam",
    "website" : "www.sunbeaminfo.in"
    "emps":[
        "emp1":{
    "id" : 2,
    "name":"pratik",
    "sal":3000,
    "comm": 200
}, "emp2":{
    "id" : 3,
    "name":"Onkar",
    "sal":4000,
}]
}

print()
System.out.println()
void sum(int num1,int num2);
```

## JSON types

```
- JSON is a part of JavaScript Language
- Number -> 123,123.45,-123
- String -> "value1","value2"
- Boolen -> true,false
- Object -> {}
- array -> []
- null
```

## Introduction to MongoDB

```
- Mongodb follows  a client server architecture
- db files are stored in .wt format.
```

```
//list all the present databases
show databases;

//create a database
use classwork;

//to see all existing collections(corresponds to Tables in RDBMS)
show collections;

db.people.insert({
    "name":"James",
    'email':'james@bond.com'
});

// db is a keyword that points to the current database.

db.people.insert({
name:'rohan',
email:'roha@sunbeaminfo.com',
mobile:'8983049388'
});
//in mongo giving the single quotes for keys is optional

db.people.insert({
name:'pratik',
email:'pratik@sunbeaminfo.com',
mobile:'1234567890',
addr:'karad'
});

//db.collection.insert();
//db.collection.insertOne();
```

```
//db.collection.insertMany();

db.people.insertOne({
name:'onkar',
email:'onkar@sunbeaminfo.com',
mobile:'1234567891',
addr:'kolhapur'
});

db.people.insertMany([
    {
        name:'yogesh',
        email:'yogesh@sunbeaminfo.com'
    },{
        name:'shubham',
        email:'shubham@sunbeaminfo.com'
    },{
        name:'pradnya',
        email:'pradnya@sunbeaminfo.com'
    }
]);

db.people.insertOne({
    _id:1002,
    name:'Nilesh',
    email:'nilesh@sunbeaminfo.com'
});

//_id is unique and is created by mongoshell
```

## Find

```
// to look for the data in collection
//db.collection.find();
//db.collection.find({criteria});
//db.collection.find({criteria},{projection});

db.people.find();

//copy paste the data from empdept.js

show collections;
db.emp.find();
db.dept.find();

//select ename,sal,comm from emp
db.emp.find({},{
    ename:1,
    sal:1,
    comm:1
});
```

```
//the above is called as inclusion projection

db.emp.find({},{
    _id:0,
    mgr:0,
    job:0,
    deptno:0
});
//the above is called as exclusion projection


db.emp.find({},{
    ename:1,
    sal:1,
    job:0,
    comm:0
});
//error inclusion and exclusion are not allowed at same time.

db.emp.find({},{
    _id:0,
    ename:1,
    sal:1
});
//cannot mix inclusion and exclusion except _id
```

## Mongo operators

```
1. Relational operators
    $eq,$ne,$gt,$lt,$gte,$lte,$in,$nin
2. Logical operators
    $and,$or,$nor
```

```
//select * from emp where ename='KING';
//db.collection.find({criteria});
db.emp.find({
    ename:'KING'
});

//select * from emp where sal>2000;
db.emp.find({
sal:{
    $gt:2000
}
});

//select * from emp where deptno!=30;
db.emp.find({
    deptno:{
```

```
        $ne:30
    }
});

//select * from emp where deptno=20 and job='ANALYST';
db.emp.find({
    $and : [
        {deptno:20},
        {job:'ANALYST'}
    ]
});

//HOMEWORK
//select * from emp where sal<2000 or job='PRESIDENT'

//select * from emp where ename IN('SMITH','JONES','CLARK')
db.emp.find({
    ename : {
        $in:['SMITH','JONES','CLARK']
    }
});

db.emp.find().pretty();

//order by sal
db.emp.find().sort({sal:1}); // Ascending
db.emp.find().sort({sal:-1}); // Desc

//order by deptno and job
db.emp.find().sort({
deptno:1,
job:1
});

//limit
db.emp.find().limit(3);
db.emp.find().skip(2).limit(1);
```

## Delete

```
//db.collection.remove({criteria});
//db.collection.deleteOne({criteria});
//db.collection.deleteMany({criteria});

db.people.find();
db.people.remove({
    email:'james@bond.com'
});

//delete one emp out of all having mobile
db.people.deleteOne({
```

```
    mobile:{
        $exists : true
    }
});

//delete all emp having mobile
db.people.deleteMany({
    mobile:{
        $exists : true
    }
});

db.people.remove({});
show collections;
db.people.drop();
show collections;
```

## Update

```
//db.collection.update({criteria},{new obj});
//db.collection.updateOne({criteria},{changes});
//db.collection.updateMany({criteria},{changes});

db.emp.insert({
_id:1,
ename:'JOHN',
sal:2000,
job:'CLERK'
});

db.emp.insert({
_id:2,
ename:'ROCK',
sal:3000,
job:'ANALYST'
});

db.emp.insert({
_id:3,
ename:'JACK',
sal:4000,
job:'MANAGER'
});

//update emp set sal=5000 where ename is JOHN
db.emp.update({ename:'JOHN'},{sal:5000});
// here old object is replaced by new object

//update emp set sal=5000 where ename is ROCK
db.emp.update({ename:'ROCK'},{
$set:{sal:5000}
```

```
});
```

## UPSERT

```
- It is a combination of Update and Insert
```

```
//db.collection.update({criteria},{new obj},upsert);

db.emp.update({
    _id:4,
    ename:'DANNY'
},{$set:{
sal:2000,
deptno:40
}
});

db.emp.update({
    _id:4,
    ename:'DANNY'
},{$set:{
sal:2000,
deptno:40
}
},false);


db.emp.update({
    _id:4,
    ename:'DANNY'
},{$set:{
sal:2000,
deptno:40
}
},true);
```

## Mongo Indexes

```
Simple
Composite
Unique
TTL -> Time To Live
GeoSpatial
```

```
//create index on mgr
db.emp.createIndex({mgr:1});

//create index on deptno and job
db.emp.createIndex({deptno:1,job:1});

//create a unique index on ename
db.emp.createIndex({ename:1},{unique:true});

//to display all indexes
db.emp.getIndexes();
db.emp.dropIndex({mgr:1});
```

## Bigdata Introduction

```
https://www.youtube.com/watch?v=BxwpqnQ6BgQ
```