MySQL - RDBMS

Agenda

- Module Overview
- DBMS vs RDBMS
- SQL
- Getting started
- Create Table
- Insert records
- Select records

Module Overview

- Syllabus
  - RDBMS (MySQL)
  - NoSQL Introduction (Mongo)
- Evaluation
  - Theory: 40 marks MCQ (CCEE)
  - Lab: 40 marks - SQL queries & PSM
    - 15-Oct / 16-Oct
  - Internals: 20 marks - Will be updated by CoCo.
- Module plan
  - C-DAC Source Book -- 72 hrs (Theory 36 hrs + Lab 36 hrs).
  - Sunbeam Database Technologies
    - 10 days (2 weeks) - 4-Oct to 16-Oct.
    - Theory: 40 hrs
      - 9.00 am to 1.30 pm.
      - Break: 11.00 am to 11.30 am.
      - Q & A: 8.45 am, 1.30 pm, 3.30 pm.
    - Lab: 36 hrs.
      - 4.00 pm to 7.00 pm.

DBMS vs RDBMS

SQL

MySQL

Installation

- Server = mysqld.exe
  - C:\Program Files\MySQL\MySQL Server 8.0\bin
- Client = mysql.exe
  - C:\Program Files\MySQL\MySQL Server 8.0\bin
- MySQL data directory/folder

    - C:\ProgramData\MySQL\MySQL Server 8.0
- Default user of MySQL = "root" (administrator).
    - Password is given during installation = "manager".
- Set PATH.
    - Windows explorer.
    - This PC (right click) --> Properties --> Advanced System Settings --> Advanced --> Environment Variables
        - User Variables --> PATH --> EDIT
        - Click New --> Add MySQL PATH at the end "C:\Program Files\MySQL\MySQL Server 8.0\bin".
- Open "Command Prompt"
    - cmd> mysql --version

Getting started

- step 1. Login with "root" user.
    - cmd> mysql -u root -p
        - Password: manager
- step 2. Create a new user.
    - mysql> CREATE USER sunbeam@localhost IDENTIFIED BY 'sunbeam';
- step 3. Create a new database/schema.
    - mysql> CREATE DATABASE classwork;
    - mysql> SHOW DATABASES;
- step 4. Give all permissions to the new user on the new database.
    - mysql> GRANT ALL PRIVILEGES ON classwork.* TO sunbeam@localhost;
    - mysql> FLUSH PRIVILEGES;
- step 5. mysql> EXIT
- step 1. Login with new user and password on MySQL CLI.
    - cmd> mysql -u sunbeam -p
        - Password: sunbeam
- step 2. Execute queries -- DDL, DML, DQL, ...
    - mysql> SHOW DATABASES;
    - mysql> SELECT USER(), DATABASE();
    - mysql> USE classwork;
    - mysql> SELECT USER(), DATABASE();
    - mysql> SHOW TABLES;
- MySQL screen clear.
    - mysql> ! cls

- CREATE TABLE
  - CREATE TABLE tablename (col1 DATATYPE, col2 DATATYPE, col3 DATATYPE, ...);
- INSERT
  - INSERT INTO tablename VALUES (v1, v2, v3, ...);
  - INSERT INTO tablename VALUES (v1, v2, v3, ...), (v1, v2, v3, ...), (v1, v2, v3, ...), ..;
- SELECT
  - SELECT * FROM tablename;
    - means All columns.

```
SHOW TABLES;

CREATE TABLE stud(id INT, name CHAR(20), marks DOUBLE);

SHOW TABLES;

DESCRIBE stud;

INSERT INTO stud VALUES (1, 'Soham', 98.20);

INSERT INTO stud VALUES (2, 'Sakshi', 97.40);

INSERT INTO stud VALUES (3, 'Prisha', 99.30), (4, 'Madhura', 96.29), (5, 'Om',
97.45);

SELECT * FROM stud;

INSERT INTO stud(name, marks, id) VALUES ('Pratham', 95.39, 6);
INSERT INTO stud(id, name) VALUES (7, 'Vedant');

SELECT * FROM stud;
```

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

SHOW TABLES;

SELECT * FROM students;

INSERT INTO students SELECT * FROM stud;

SELECT * FROM students;
```

```
DROP TABLE students;
```

```
SHOW TABLES;
```

MySQL - RDBMS

Agenda

- Logical vs Physical Layout
- MySQL data types
- CHAR vs VARCHAR vs TEXT
- SQL scripts
- DQL - SELECT
    - Projection
    - Computed Columns
    - DISTINCT
    - LIMIT
    - ORDER BY
    - WHERE
        - Relational Operators
        - NULL Operators
        - Logical Operators

DBMS

- DBMS -- Database Management (CRUD)
    - Traditional databases - File based databases
    - Relational databases
    - NoSQL databases

Logical vs Physical Layout

MySQL data types

```
* CREATE TABLE stud(id INT, name CHAR(20), marks DOUBLE);
    * id is by default signed.
* CREATE TABLE stud(id INT UNSIGNED, name CHAR(20), marks DECIMAL(5,2));
    * id is now unsigned.
    * DECIMAL(5,2) -- total 5 digits and 2 digits after decimal point.
```

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

DESCRIBE temp;

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

INSERT INTO temp VALUES('xy', 'xy', 'xy');

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

```
INSERT INTO temp VALUES('pqrst', 'pqr', 'pqr');
-- error

INSERT INTO temp VALUES('pqr', 'pqrst', 'pqr');
-- error

SELECT * FROM temp;
```

CHAR vs VARCHAR vs TEXT

SQL scripts

```
USE classwork;

SELECT USER(), DATABASE();
-- sunbeam@localhost, classwork

SOURCE D:/classwork-db.sql

SHOW TABLES;

SELECT * FROM books;

SELECT * FROM dept;
```

DQL - SELECT

- Projection -- Display only given columns.

```
SELECT * FROM dept;
-- * all columns

SELECT deptno, dname FROM dept;

SELECT * FROM emp;

SELECT empno, ename, sal FROM emp;

SELECT sal, ename, deptno FROM emp;

CREATE TABLE newemp(id INT, name CHAR(30), sal DOUBLE);

INSERT INTO newemp SELECT * FROM emp;
-- error

INSERT INTO newemp SELECT empno, ename, sal FROM emp;

SELECT * FROM newemp;
```

```
INSERT INTO newemp(id, name, sal) SELECT empno, ename, sal FROM emp;

SELECT * FROM newemp;
```

```
-- display ename, sal, da=sal*0.5, gs=sal+sal*0.5
SELECT ename, sal, sal*0.5, sal+sal*0.5 FROM emp;
-- computed columns or pseudo columns.

SELECT ename AS name, sal, sal * 0.5 AS da, sal + sal * 0.5 AS gs FROM emp;
-- column alias

SELECT ename name, sal, sal * 0.5 da, sal + sal * 0.5 gs FROM emp;
-- writing AS is optional

SELECT ename name, sal, sal * 0.5 da, sal + sal * 0.5 gross sal FROM emp;
-- error

SELECT ename name, sal, sal * 0.5 da, sal + sal * 0.5 `gross sal` FROM emp;
```

```
-- display ename, deptno and dname (10=ACCOUNTING, 20=RESEARCH, 30=SALES).
SELECT ename, deptno FROM emp;

SELECT ename, deptno, CASE
WHEN deptno = 10 THEN 'ACCOUTING'
WHEN deptno = 20 THEN 'RESEARCH'
WHEN deptno = 30 THEN 'SALES'
ELSE 'UNKNOWN'
END AS deptname
FROM emp;
```

```
SELECT DISTINCT job FROM emp;

SELECT DISTINCT deptno FROM emp;

SELECT deptno, job FROM emp;
-- 10 --> CLERK, MANAGER, PRESIDENT
-- 20 --> CLERK, ANALYST, MANAGER
-- 30 --> CLERK, MANAGER, SALESMAN

SELECT DISTINCT deptno, job FROM emp;
-- displays unique combination of deptno & job.
```

```
DESCRIBE emp;
```

```
SELECT * FROM emp;
-- show all rows

SELECT * FROM emp LIMIT 5;
-- show first 5 rows.

SELECT * FROM emp LIMIT 10;
-- show first 10 rows.

SELECT ename, sal FROM emp LIMIT 10;
-- show ename & sal of first 10 rows.

SELECT ename, sal FROM emp LIMIT 5;
-- show ename & sal of first 5 rows.

SELECT ename, sal FROM emp LIMIT 3, 5;
-- show ename & sal of 5 rows after first 3 rows.
```

```
SELECT * FROM emp;

SELECT * FROM emp ORDER BY sal DESC;

SELECT * FROM emp ORDER BY deptno ASC;

SELECT * FROM emp ORDER BY hire;
-- default sort = ASC.

SELECT * FROM emp ORDER BY deptno ASC, job ASC;
-- sort on multiple columns.

SELECT * FROM emp ORDER BY deptno DESC, job, ename;
-- sort on deptno (desc), job (asc), ename (asc)
```

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

-- display emp whose name is last in alphabetically.
SELECT * FROM emp ORDER BY ename DESC LIMIT 1;

SELECT * FROM emp ORDER BY comm;

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

-- display emp with third lowest sal.
SELECT * FROM emp ORDER BY sal LIMIT 2,1;

-- display emp with second highest sal.
```

```sql
SELECT * FROM emp ORDER BY sal DESC;

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

-- sort emp on da=sal*0.5
SELECT ename, sal, sal*0.5 da FROM emp;

SELECT ename, sal, sal*0.5 da FROM emp
ORDER BY sal*0.5;
-- order by expr

SELECT ename, sal, sal*0.5 da FROM emp
ORDER BY da;
-- order by column alias

SELECT ename, sal, sal*0.5 da FROM emp
ORDER BY 3;
-- order by column numer (in projection)
```

```sql
-- display emps of dept 30
SELECT * FROM emp WHERE deptno = 30;

-- display all emps with sal > 2000.0
SELECT * FROM emp WHERE sal > 2000.0;

-- display all ANALYST.
SELECT * FROM emp WHERE job = 'ANALYST';

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

```sql
-- display emps in sal range from 1000 to 2000.
-- sal >= 1000 && sal <= 2000
SELECT * FROM emp WHERE sal >= 1000 AND sal <= 2000;

-- display ANALYSTs and MANAGERs.
-- job == 'ANALYST' || job == 'MANAGER';
SELECT * FROM emp WHERE job = 'ANALYST' OR job = 'MANAGER';

-- display emps who are not salesman.
SELECT * FROM emp WHERE job != 'SALESMAN';
SELECT * FROM emp WHERE job <> 'SALESMAN';

-- !(condition)
SELECT * FROM emp WHERE NOT job = 'SALESMAN';
```

```
-- display all emps hired in 1982.
-- '1-1-1982' to '31-12-1982'
SELECT * FROM emp WHERE hire >= '1982-01-01' AND hire <= '1982-12-31';
```

```
-- display all emps whose comm is null.
SELECT * FROM emp WHERE comm = NULL;
-- relational operators cannot be used with NULL.
-- NULL is used with special operators.

SELECT * FROM emp WHERE comm IS NULL;

SELECT * FROM emp WHERE comm <=> NULL;

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

SELECT * FROM emp WHERE NOT (comm IS NULL);
```

MySQL - RDBMS

Agenda

- DQL - SELECT
  - WHERE clause
    - IN operator
    - BETWEEN operator
    - LIKE operator
- DML - UPDATE
- DML - DELETE
- DDL - TRUNCATE
- DDL - DROP
- HELP
- DUAL table
- SQL Functions
  - String Functions

WHERE clause

- terminal> mysql -u sunbeam -psunbeam classwork
- IN operator
  - Similar to Logical OR for equality checking with multiple values (for same column).
  - NOT IN operator -- inverse of IN operator

```
SELECT USER(), DATABASE();
-- | sunbeam@localhost | classwork  |

-- display all MANAGERS and emps from dept 10.
SELECT * FROM emp WHERE job = 'MANAGER' OR deptno = 10;

-- display all emps whose sal is less than 1000 or sal is more than 3500.
SELECT * FROM emp WHERE sal < 1000 OR sal > 3500;

-- display all analysts and managers.
SELECT * FROM emp WHERE job = 'ANALYST' OR job = 'MANAGER';

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

-- display emps whose names are JAMES, KING, MARTIN, FORD.
SELECT * FROM emp WHERE ename = 'JAMES' OR ename = 'KING' OR ename = 'MARTIN' OR
ename = 'FORD';

SELECT * FROM emp WHERE ename IN ('JAMES', 'KING', 'MARTIN', 'FORD');

-- display all emps whose are not salesman or manager.
SELECT * FROM emp WHERE job NOT IN ('SALESMAN', 'MANAGER');
```

```
SELECT * FROM emp WHERE NOT (job IN ('SALESMAN', 'MANAGER'));

SELECT * FROM emp WHERE NOT (job = 'SALESMAN' OR job = 'MANAGER');
```

- BETWEEN operator
  - range check (including both ends).
  - col BETWEEN start AND end
    - col >= start AND col <= end.

```
-- display manager of dept 20.
SELECT * FROM emp WHERE job = 'MANAGER' AND deptno = 20;

-- display all emps in dept 30 whose sal is less than 1500.
SELECT * FROM emp WHERE deptno = 30 AND sal < 1500;

-- display all emps in sal range 1500 to 3000.
SELECT * FROM emp WHERE sal >= 1500 AND sal <= 3000;

SELECT * FROM emp WHERE sal BETWEEN 1500 AND 3000;

-- display all emps hired in year 1982.
SELECT * FROM emp WHERE hire >= '1982-01-01' AND hire <= '1982-12-31';

SELECT * FROM emp WHERE hire BETWEEN '1982-01-01' AND '1982-12-31';
```

- Paperwork
  - ADAM
  - B
  - BLAKE
  - CLARK
  - J
  - JAMES
  - KING

```
INSERT INTO emp(ename) VALUES ('B'), ('J');

SELECT ename FROM emp ORDER BY ename;

-- display all emps whose names start with 'B' to 'J';
SELECT ename FROM emp WHERE ename BETWEEN 'B' AND 'J';
-- JAMES & JONES are not displayed because alphabetically they come after J.

SELECT ename FROM emp WHERE ename BETWEEN 'B' AND 'K';
-- will display all names starting from B to J + will also display "K" (if
present).

SELECT ename FROM emp WHERE ename BETWEEN 'B' AND 'K' AND ename != 'K';
```

```
-- will display all names starting from B to J

-- display all emps whose sal is not in range 1000 to 2000.
SELECT * FROM emp WHERE sal NOT BETWEEN 1000 AND 2000;

-- display all emps whose sal is not in range 1500 to 3000.
SELECT * FROM emp WHERE sal NOT BETWEEN 1500 AND 3000;

-- display all emps between B to K and T to Z;
SELECT * FROM emp WHERE
ename BETWEEN 'B' AND 'K'
OR
ename BETWEEN 'T' AND 'Z';
```

- LIKE operator.
  - Used with strings for finding similar records.
  - Wildcard character to give pattern.
    - % : Any number of any char or Empty.
    - _ : Single occurrence of any char.
  - NOT LIKE : inverse of LIKE

```
-- find all emps whose name start with M.
SELECT * FROM emp WHERE ename LIKE 'M%';

-- find all emps whose name ends with H.
SELECT * FROM emp WHERE ename LIKE '%H';

-- find all emps whose name contain U.
SELECT * FROM emp WHERE ename LIKE '%U%';

-- find all emps whose name contains A twice.
SELECT * FROM emp WHERE ename LIKE '%A%A%';
-- ADAMS --> YES --> %=, %=D, %=MS
-- WARD --> NO
-- ANNA --> YES --> %=, %=NN, %=
-- RAAM --> YES --> %=R, %=, %=M

-- find all emps whose name is start with S to Z.
SELECT * FROM emp WHERE ename BETWEEN 'S' AND 'Z'
OR ename LIKE 'Z%';
```

```
-- display emps having 4 letter name.
SELECT * FROM emp WHERE ename LIKE '____';

-- display all emps whose name contains R on 3rd position.
SELECT * FROM emp WHERE ename LIKE '__R%';
```

```
-- display emps with 4 letter name and 3rd pos is R
SELECT * FROM emp WHERE ename LIKE '__R_';
```

- SELECT cols FROM tablename WHERE condition ORDER BY cols LIMIT n;

```
-- display emp with highest sal in range 1000 to 2000.
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'
ORDER BY sal
LIMIT 1;

-- diplay fifth lowest sal from dept 20 and 30.
SELECT sal FROM emp WHERE deptno IN (20, 30);

SELECT sal FROM emp WHERE deptno IN (20, 30)
ORDER BY sal;

SELECT DISTINCT sal FROM emp
WHERE deptno IN (20, 30)
ORDER BY sal LIMIT 4, 1;
```

DML - UPDATE

```
SHOW TABLES;

SELECT * FROM stud;

-- change marks to 95.00 for student with id 7.
UPDATE stud SET marks = 95.80 WHERE id = 7;

SELECT * FROM stud;

SELECT id,name,subject,price FROM books;

-- increase price of C Programming books by 50.
UPDATE books SET price = price + 50
WHERE subject = 'C Programming';

SELECT id,name,subject,price FROM books;

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

SELECT id,name,subject,price FROM books;
```

```
SELECT * FROM emp;

UPDATE emp SET empno=1, sal=1000, comm=NULL WHERE ename = 'B';

SELECT * FROM emp;
```

DML - DELETE

```
-- delete given rows from emp table -- ename = B / J
DELETE FROM emp WHERE ename IN ('B', 'J');

SELECT * FROM emp;

SELECT * FROM newemp;

-- delete all rows from emp table
DELETE FROM newemp;

SELECT * FROM newemp;

DESCRIBE newemp;

-- delete price of book with id 1001 -- edit/update
UPDATE books SET price = NULL WHERE id = 1001;

SELECT * FROM books;
```

DDL - TRUNCATE

- TRUNCATE is DDL query.
- TRUNCATE is to delete all rows - cannot use WHERE clause.
- Table structure is not deleted (simiar to DELETE query).

```
-- delete all books
TRUNCATE TABLE books;

SELECT * FROM books;

DESCRIBE books;
```

DDL - DROP

- DROP command can be used for dropping/deleting whole table or database.
  - DROP TABLE tablename;
  - DROP DATABASE dbname;

```
SELECT * FROM dummy;

DESCRIBE dummy;

DROP TABLE dummy;

SHOW TABLES;

DESCRIBE dummy;
-- error
```

DELETE vs TRUNCATE vs DROP

- DELETE
  - Used to delete rows (not structure).
  - All rows or as per WHERE condition.
  - DML operation
  - DML ops can be rollbacked (undo/discard) using transaction.
  - In most RDBMS, slower operation.
- TRUNCATE
  - Used to delete rows (not structure).
  - All rows.
  - DDL operation.
  - DDL ops cannot be rollbacked.
  - In most RDBMS, faster operation.
- DROP
  - Used to delete rows as well as struct.
  - DDL operation.
  - DDL ops cannot be rollbacked.
  - This is fastest operation.

HELP

```
HELP SELECT;

HELP INSERT;

HELP Functions;

HELP String Functions;

HELP UPPER;

SELECT UPPER('MySql');

SELECT SUBSTRING('SUNBEAM', 4, 2);
```

DUAL Table

- In Oracle, DUAL table.
  - DUAL table is in memory single row single column virtual table to support SELECT query syntax.
  - SELECT cols FROM tablename;

```
SELECT 2 + 4 * 5 FROM DUAL;

SELECT VERSION() FROM DUAL;

SELECT NOW() FROM DUAL;

SELECT LOWER('SunBeam') FROM DUAL;
```

- One cannot manipulate or drop DUAL table.

```
DESCRIBE DUAL;
-- error

DROP TABLE DUAL;
-- error

DELETE FROM DUAL;
-- error
```

- ANSI SQL influenced by Oracle SQL.
- DUAL table is added in ANSI standard.
- In ANSI SQL, this table is optional.

```
SELECT 2 + 4 * 5;

SELECT VERSION();

SELECT NOW();

SELECT LOWER('SunBeam');
```

SQL Functions

String Functions

```
SELECT LOWER('India');

SELECT ename, LOWER(ename) FROM emp;
```

```
SELECT LEFT('Sunbeam', 2), RIGHT('Sunbeam', 2);

SELECT ename, LEFT(ename, 2), RIGHT(ename, 2) FROM emp;

-- display all emps whose name start with B to K.
SELECT ename, LEFT(ename,1) FROM emp
WHERE LEFT(ename,1) BETWEEN 'B' AND 'K';

SELECT SUBSTRING('SUNBEAM', 2, 3);
-- UNB (+ve pos -- pos from left)

SELECT SUBSTRING('SUNBEAM', -5, 3);
-- NBE (-ve pos -- pos from right)

SELECT SUBSTRING('SUNBEAM', 4, 0);
-- len=0, means no chars after given pos - empty

SELECT SUBSTRING('SUNBEAM', 4, -2);
-- len < 0, means no chars after given pos - empty

-- print chars 2-5 for all emp names.
SELECT ename, SUBSTRING(ename, 2, 4) FROM emp;

SELECT CONCAT('SUNBEAM', ' ', 'INFOTECH');

SELECT CONCAT('SUNBEAM', 2021);

SELECT CONCAT(ename, ' - ', job) FROM emp;

SELECT CONCAT(ename, ' is working in dept ', deptno, ' as ', job, '.') FROM emp;

SELECT LENGTH('    abcd  ');

SELECT TRIM('    abcd  ');

SELECT LENGTH(TRIM('    abcd  '));
-- output of TRIM('    abcd  ') is given as input to LENGTH().

SELECT LPAD('Sunbeam', 10, '*');
SELECT RPAD('Sunbeam', 10, '*');

SELECT RPAD(LPAD('Sunbeam', 10, '*'), 13, '*');
```

MySQL - RDBMS

Agenda

- SQL Functions
  - Numeric Functions
  - Date and Time Functions
  - Control Flow Functions
  - Misc/Info Functions
  - List Functions
  - NULL Functions
  - Group Functions
- GROUP BY clause
- HAVING clause

SQL Functions

Numeric Functions

```
SELECT USER(), DATABASE();
-- | sunbeam@localhost | classwork  |

HELP Numeric Functions;

SELECT POWER(2, 5);

SELECT SQRT(2);

SELECT RAND();

-- fetch rows in random order
SELECT empno, ename, sal FROM emp
ORDER BY RAND();

SELECT PI();

SELECT ROUND(3.141593, 2), ROUND(3.141593, 4);
-- 3.14, 3.1416

SELECT ROUND(314159.3, -2), ROUND(31415.93, -2);
-- 314200, 31400

SELECT ROUND(3.141593, -2);
-- 0

SELECT ROUND(7246851749, -5);

SELECT * FROM books;
```

```
SELECT id, name, ROUND(price,2) FROM books;
```

- CEIL -- nearest higher integer.
- FLOOR -- nearest lower integer.

```
SELECT CEIL(3.14), FLOOR(3.14);
-- 4, 3

SELECT CEIL(-3.14), FLOOR(-3.14);
-- -3, -4
```

Date and Time Functions

- DATE -- '1000-01-01' to '9999-12-31'
- TIME -- '-838:59:59' to '838:59:59'
- DATETIME -- '1000-01-01 00:00:00.000000' to '9999-12-31 23:59:59.999999'
- TIMESTAMP -- '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC
- YEAR -- '1901' to '2155'

```
HELP Date and Time Functions;

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

SELECT DATE('2000-05-24 14:47:20'), TIME('2000-05-24 14:47:20');

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

SELECT DATE_ADD(NOW(), INTERVAL 4 DAY);

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

SELECT DATEDIFF(NOW(), '1983-09-28');
-- return number of days

SELECT TIMESTAMPDIFF(YEAR, '1983-09-28', NOW());
-- return number of years

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

SELECT ename, hire, TIMESTAMPDIFF(YEAR, hire, NOW()) exp_yrs, TIMESTAMPDIFF(MONTH,
hire, NOW()) % 12 exp_mons FROM emp;

SELECT DATE_FORMAT(NOW(), '%d-%b-%Y');
-- 07-Oct-2021

SELECT DAY(NOW()), MONTH(NOW()), YEAR(NOW()), HOUR(NOW()), MINUTE(NOW()),
```

```
SECOND(NOW()), WEEKDAY(NOW());

-- find all emps hired in 1982
SELECT * FROM emp WHERE YEAR(hire) = 1982;

-- MySQL standard date format: 'yyyy-mm-dd'
-- input date: 'dd/mm/yyyy' -- ??
SET @str = '28/09/2021';
SELECT STR_TO_DATE(@str, '%d/%m/%Y');
```

Control Functions

- IF(condition, expr_if_true, expr_if_false).

```
HELP IF FUNCTION;

-- display ename, sal and category
-- category = RICH if sal > 2500
-- category = POOR if sal <= 2500
SELECT ename, sal, IF(sal > 2500, 'RICH', 'POOR') AS category FROM emp;

-- print number is +ve or -ve or zero.
SET @num = 2;
-- MySQL user-defined variable -- session scope.
-- when MySQL CLI exit, variable will be destroyed

SELECT @num;

SELECT IF(@num > 0, '+ve', IF(@num < 0, '-ve', 'zero'));
```

List Functions

```
SELECT CONCAT('A', 12, 'B', 34.45);

SELECT GREATEST(23, 98, 53, 67);

SELECT LEAST(23, 98, 53, 67);

SELECT name, price, LEAST(price,700) FROM books;

SELECT GREATEST('AEROPLANE', 'CAR');

SELECT LEAST('AEROPLANE', NULL, 'CAR');

SELECT CONCAT('A', 12, NULL, 'B', 34.45);
```

Misc Functions

```
SELECT VERSION();

SELECT SYSDATE();

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

NULL Functions

- NULL is special value in RDBMS.
- It is irrespective of data type.
- NULL is not 0, 0.0, '\0', 'NULL'.
- NULL represent missing/absent/empty value.

```
SELECT COALESCE(NULL, NULL, 12, 'Nilesh');
-- return = first non-null value.

SELECT COALESCE(12.34, NULL, 'Nilesh');

-- display comm of emp and if no comm then display sal.
SELECT ename, comm, sal, COALESCE(comm, sal) FROM emp;

SELECT ename, comm, sal, IF(comm IS NULL, sal, comm) FROM emp;

SELECT ename, comm, sal, IFNULL(comm, sal) FROM emp;
-- if arg1 == NULL, then result = arg2, else arg1.

SELECT ename, sal, NULLIF(sal,3000.0) FROM emp;
-- if arg1 == arg2, then result = NULL, else arg1.

SELECT IFNULL(NULL, 'Hello');
```

Group Functions

- Single Row Functions:
  - "n" Input Rows --> "n" Output Rows
  - Function execute once for each row.
- Group Functions/Multi Row Functions/Aggregate Functions
  - "n" Input Rows --> "1" Output Row
  - Aggregate value
    - COUNT(), SUM(), AVG(), MAX(), MIN()
    - STDEV(), COR(), ...

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

SELECT COUNT(comm), SUM(comm), AVG(comm), MAX(comm), MIN(comm) FROM emp;
-- NULL values are ignored by Group Functions.
```

```
-- display max income and min income from emp.
-- income = sal + comm
SELECT ename, sal, comm, sal + IFNULL(comm,0) AS income FROM emp;

SELECT MAX(sal + IFNULL(comm,0)), MIN(sal + IFNULL(comm,0)) FROM emp;
```

- GREATEST vs MAX and LEAST vs MIN
  - GREATEST/LEAST -- single row function.
  - MAX/MIN -- group/aggregate function.
  - GREATEST/LEAST -- operate on multiple values from the same row.
  - MAX/MIN -- operate on multiple values in different rows (given column).
  - GREATEST/LEAST -- list function (multiple args).
  - MAX/MIN -- single arg (column name)
  - GREATEST/LEAST -- if any arg is NULL, result is NULL.
  - MAX/MIN -- if any row has NULL value in given column, that will be ignored.

LIMITATIONS OF GROUP FUNCTIONS

```
SET @@sql_mode='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION';

SELECT ename, MAX(sal) FROM emp;
-- error: cannot select any column with group fn.

SELECT LOWER(ename), MAX(sal) FROM emp;
-- error: cannot select single row fn with group fn.

SELECT * FROM emp WHERE sal = MAX(sal);
-- error: cannot use group fn in WHERE clause

SELECT SUM(MAX(sal)) FROM emp;
-- error: cannot nest group fn in each other.
```

- To set sql_mode permanently.
  - step 1: Run Notepad -- "Run as Administrator".
  - step 2: Open my.ini from C:\ProgramData\MySQL\MySQL Server 8.0.
  - step 3: Under [mysqld] change sql_mode.
    - sql-mode=ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION
  - step 4: Restart MySQL server (or restart computer).

DQL - SELECT

GROUP BY clause

- By default GROUP functions work on all the rows.
- With GROUP BY we can use GROUP functions on group of rows.

```
SELECT deptno, COUNT(sal), SUM(sal), AVG(sal), MAX(sal), MIN(sal) FROM emp
GROUP BY deptno;

SELECT job, COUNT(sal), SUM(sal), AVG(sal), MAX(sal), MIN(sal) FROM emp
GROUP BY job;

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

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

```
SELECT empno, ename, deptno, job FROM emp
ORDER BY deptno, job;

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

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

-- deptno   job     count
--  10      C       1
--  10      M       1
--  10      P       1
--  20      A       2
--  20      C       2
--  20      M       1
--  30      C       1
--  30      M       1
--  30      S       4
```

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

```
-- deptwise total sal
SELECT deptno, SUM(sal) FROM emp GROUP BY deptno;

SELECT SUM(sal) FROM emp GROUP BY deptno;
-- it is not mandetory to project grouped column.
-- however output will be meaningless.

SELECT deptno, SUM(sal) FROM emp;
-- error: cannot select column with group fn.
```

HAVING clause

- Must be used with GROUP BY clause only.
- Mainly used to apply condition on aggregate values/results.
- HAVING clause vs WHERE clause
    - WHERE clause: evaluated for each row.
    - HAVING clause: evaluated for each group.
    - WHERE clause: can be used with column, single row fn, but not group fn.
    - HAVING clause: can be used with grouped column or group fn, but not on other columns.

```
-- display deptno in which total sal is more than 9000.
SELECT deptno, SUM(sal) FROM emp
GROUP BY deptno
HAVING SUM(sal) > 9000;

-- display jobs for which avg sal is more than 2500.
SELECT job, AVG(sal) FROM emp
GROUP BY job
HAVING AVG(sal) > 2500;

-- display max sal for each job for emps in deptno 10 and 20.
SELECT * FROM emp WHERE deptno IN (10,20);

SELECT job, MAX(sal) FROM emp
WHERE deptno IN (10,20)
GROUP BY job;

-- display max sal for each job for emps in deptno 10 and 20. display max sal only
if it is more than 2500.

SELECT job, MAX(sal) FROM emp
WHERE deptno IN (10,20)
GROUP BY job
HAVING MAX(sal) > 2500;
```

```
-- find avg sal for deptno 10 and 20.
SELECT deptno, AVG(sal) FROM emp
WHERE deptno IN (10, 20)
GROUP BY deptno;
-- more efficient

SELECT deptno, AVG(sal) FROM emp
GROUP BY deptno
HAVING deptno IN (10, 20);
-- less efficient
```

```
-- find the dept that spends max on emp sals.
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;
```

```
-- find the jobs which have lowest avg sal.
SELECT job, AVG(sal) FROM emp
GROUP BY job
ORDER BY AVG(sal)
LIMIT 1;

-- find the jobs which have lowest avg income.
SELECT job, AVG(sal + IFNULL(comm,0.0)) FROM emp
GROUP BY job
ORDER BY AVG(sal + IFNULL(comm,0.0))
LIMIT 1;
```

MySQL - RDBMS

Agenda

- Joins

Case sensitive string comparision

```
SELECT 'SunBeam' = 'SUNBEAM';

SELECT BINARY 'SunBeam' = BINARY 'SUNBEAM';

SELECT BINARY 'SUNBEAM' = BINARY 'SUNBEAM';

SELECT * FROM emp WHERE ename = 'King';
-- case insensitive search

SELECT * FROM emp WHERE BINARY ename = BINARY 'King';
-- case sensitive search -- do not match

SELECT * FROM emp WHERE BINARY ename = BINARY 'KING';
-- case sensitive search -- match
```

Joins

```
USE classwork;

SELECT USER(), DATABASE();
-- sunbeam@localhost, classwork

SOURCE D:/sep21/DAC/dbt/db/joins.sql

SHOW TABLES;

SELECT * FROM emps;

SELECT * FROM depts;

SELECT * FROM addr;

SELECT * FROM meeting;

SELECT * FROM emp_meeting;
```

Cross Joins

```
// for loop
for(int i=0; i<emp.length; i++) {
    Emp e = emp[i];
    for(int j=0; j<dept.length; j++) {
        Dept d = dept[j];
        System.out.println(e.ename + " -- " + d.dname);
    }
}
```

```
// for-each loop
for(Emp e:emp) {
    for(Dept d:dept) {
        System.out.println(e.ename + " -- " + d.dname);
    }
}
```

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

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

```
SELECT e.ename, d.dname FROM emps AS e
CROSS JOIN depts AS d;
-- can use AS keyword for table alias

SELECT emps.ename, depts.dname FROM emps
CROSS JOIN depts;
-- using alias is not mandetory, we can directly use table name

SELECT ename, dname FROM emps
CROSS JOIN depts;
-- if column names are different in both tables, writing alias/tablename is
optional

SELECT ename, dname, deptno FROM emps
CROSS JOIN depts;
-- ERROR: Column 'deptno' in field list is ambiguous.

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

Inner Join

- Joining two tables (Getting data from two tables based on some condition).

- Obviously the table must be related by some way (some column).
    - One DEPT can have Many EMP.
    - Many EMP can be in One DEPT.
    - This relation is established with "deptno" column in depts and in emps.

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

-- display ename and names of dept in which he is not working
SELECT e.ename, d.dname FROM emps e
INNER JOIN depts d ON e.deptno != d.deptno;
```

Equi-join

- When in any join query condition is of equality, it is referred as "equi-join".

Non-equi-join

- When in any join query condition is of non-equality(<, >, <=, >=, !=), it is referred as "non-equi-join".

Left Outer Join

- Left Join = Intersection + Extra rows from Left table

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

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

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

Right Outer Join

- Right Join = Intersection + Extra rows from Right table

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

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

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

Full Outer Join

- Full Join = Intersection + Extra rows from Left table + Extra rows from Right table
- Full Outer Join is not supported in MySQL. It can work well in Oracle, MS-SQL, ...

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

Set Operators

- Used to combine results of two queries (if output contains same number of columns).

```
(SELECT dname AS name FROM depts)
UNION ALL
(SELECT ename FROM emps);
```

```
(SELECT sal FROM emp)
UNION ALL
(SELECT price FROM books);
```

```
(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);
```

```
(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);
-- simulation of full outer join in MySQL
```

Self Join

```
-- print ename and his manager name.
SELECT e.ename, m.ename AS mname FROM emps e
```

```
INNER JOIN emps m ON e.mgr = m.empno;

-- print ename and his manager name.
SELECT e.ename, m.ename AS mname FROM emps e
LEFT JOIN emps m ON e.mgr = m.empno;
```

Joins Practice

```
-- display ename, emp's dname and emp's dist.
SELECT e.ename, d.dname FROM emps e
INNER JOIN depts d ON e.deptno = d.deptno;

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

```
-- display ename and his meeting topics.
SELECT * FROM emps;

SELECT * FROM meeting;

SELECT * FROM emp_meeting;

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

```
-- emps are travelling from their home town to attend few meetings. Display name
of emp and meeting topic and from where he is travelling.

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

-- emps are representing their depts in few meetings. Display name of emp and
meeting topic and their dept.
SELECT e.ename, m.topic, d.dname
FROM emp_meeting em
INNER JOIN emps e ON em.empno = e.empno
INNER JOIN meeting m ON em.meetno = m.meetno
LEFT JOIN depts d ON e.deptno = d.deptno;
```

```
-- print dname and number (count) of emps in that dept.
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 number of meetings in desc order of meeting count.
SELECT e.ename, m.topic FROM emp_meeting em
INNER JOIN emps e ON em.empno = e.empno
INNER JOIN meeting m ON em.meetno = m.meetno;

SELECT em.empno, COUNT(em.meetno)
FROM emp_meeting em
GROUP BY em.empno;

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

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

```
-- display all emps in DEV dept.
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
INNER JOIN depts d ON e.deptno = d.deptno
WHERE d.dname = 'DEV';
```

```
SELECT columns FROM table1
xxx JOIN table2 ON condition
xxx JOIN table3 ON condition ...
WHERE condition
GROUP BY column
HAVING condition
```

```
ORDER BY column
LIMIT n;
```

Non-standard joins

```
-- display ename and dname.
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;
-- by default join is INNER (in MySQL).

SELECT e.ename, d.dname FROM emps e
CROSS JOIN depts d ON e.deptno = d.deptno;
-- In MySQL, we can apply condition on CROSS JOIN

SELECT e.ename, d.dname FROM emps e
CROSS JOIN depts d WHERE e.deptno = d.deptno;
-- You may use WHERE clause with CROSS JOIN
-- However choose INNER JOIN if applicable.

SELECT e.ename, d.dname FROM emps e, depts d
WHERE e.deptno = d.deptno;
-- Join without JOIN keyword is old-style join.

SELECT e.ename, d.dname FROM emps e
INNER JOIN depts d USING (deptno);
-- joined columns from both tables have SAME name
-- the condition can be given using USING keyword
-- USING (colname) --> t1.colname = t2.colname;
-- This is always eqi-join.
-- This works only in MySQL.

SELECT e.ename, d.dname FROM emps e
NATURAL JOIN depts d;
-- num of joined columns = 1 (same name)
-- NATURAL JOIN = Implicit Join Condition
-- Equality Check of Columns with Same Name in Both tables.
-- In this example
--  NATURAL JOIN: ON e.deptno = d.deptno.

-- display all possible depts for Amit & Nilesh.
SELECT e.ename, d.dname FROM emps e
CROSS JOIN depts d WHERE e.ename IN ('AMIT', 'NILESH');
```

Natural Join

- table1: a, b, c, d
- table2: a, b, x, y

- table1 NATURAL JOIN table2 --> ON t1.a = t2.a AND t1.b = t2.b;

MySQL - RDBMS

Agenda

- Sub-queries
- Views

Q & A

```
SELECT deptno, SUM(sal) FROM emp
WHERE job != 'MANAGER'
GROUP BY deptno;
```

```
UPDATE dept d
INNER JOIN emp e ON e.deptno = d.deptno
SET d.dname = 'ACCOUNTING'
WHERE e.ename = 'KING';

SELECT * FROM dept;
```

```
SELECT d.deptno, d.dname, SUM(e.sal) FROM emp e
INNER JOIN dept d ON e.deptno = d.deptno
GROUP BY d.deptno, d.dname;
```

Assignment 6

```
SELECT o.onum, c.cname FROM orders o
INNER JOIN customers c ON o.cnum = c.cnum;
```

```
SELECT o.onum, c.cname, s.sname FROM orders o
INNER JOIN customers c ON o.cnum = c.cnum
INNER JOIN salespeople s ON o.snum = s.snum;
```

```
-- wrong query
SELECT o.onum, c.cname, s.sname FROM orders o
INNER JOIN customers c ON o.cnum = c.cnum
INNER JOIN salespeople s ON c.snum = s.snum;
```

```
SELECT c.cname, s.sname, s.comm FROM customers c
INNER JOIN salespeople s ON c.snum = s.snum
WHERE s.comm > 0.12;
```

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

```
SELECT s1.snum snum1, s1.sname sname1, s2.snum s2, s2.sname sname2 FROM
salespeople s1
INNER JOIN salespeople s2 ON s1.city = s2.city
WHERE s1.snum < s2.snum;
```

RDBMS Preparations

- Interview preparations -- SQL queries -- Joins, Sub-queries
  - Assignments
  - Assesments
  - Hackerrank -- SQL -- Easy & Medium.
- CCEE preparations -- MCQ
  - Moodle MCQ activity
    - Data entry till end of module -- After that read-only
    - Search functionality -- Topicwise, Wordwise.
- Lab exams
  - Lab Assignments & Assesments

Sub-queries

Single row sub-queries

- Return single row from inner query.

```
USE classwork;

-- display emp with max sal.

SELECT * FROM emp ORDER BY sal DESC LIMIT 1;
-- will produce partial output if multiple emps have same max sal.

SELECT * FROM emp WHERE sal = MAX(sal);
-- error: Group fns cannot be used in WHERE clause

SET @maxsal = (SELECT MAX(sal) FROM emp);
```

```
SELECT @maxsal;
SELECT * FROM emp WHERE sal = @maxsal;

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

-- display emp with second highest sal
SELECT * FROM emp ORDER BY sal DESC LIMIT 1, 1;

SET @sal2 = (SELECT DISTINCT sal FROM emp ORDER BY sal DESC LIMIT 1,1);
SELECT @sal2;
SELECT * FROM emp WHERE sal = @sal2;

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

-- display emp with third highest sal
SELECT * FROM emp WHERE sal = (SELECT DISTINCT sal FROM emp ORDER BY sal DESC
LIMIT 2,1);

-- display emps working in dept of 'KING'
SET @dno = (SELECT deptno FROM emp WHERE ename = 'KING');
SELECT * FROM emp WHERE deptno = @dno;

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';
```

Multi-row sub-queries

- Inner query returns multiple rows.
- They can be compared using ANY, ALL or IN operator.

```
-- find all emps having sal more than all salesman.
SELECT sal FROM emp WHERE job = 'SALESMAN';

SELECT MAX(sal) FROM emp WHERE job = 'SALESMAN';

SELECT * FROM emp WHERE sal > (SELECT MAX(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 > 1250 AND sal > 1500)
```

```
-- find emp with sal less than sal of "any" emp in deptno=20
SELECT sal FROM emp WHERE deptno = 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);
-- sal < 800 OR sal < 2975 OR sal < 3000 OR sal < 1100 OR sal < 3000.
```

```
-- display the depts which have emps.
SELECT deptno FROM emp;

SELECT * FROM dept WHERE deptno = ANY(SELECT deptno FROM emp);
-- deptno = 10 OR deptno = 20 OR deptno = 30

SELECT * FROM dept WHERE deptno IN (SELECT deptno FROM emp);
-- deptno = 10 OR deptno = 20 OR deptno = 30
```

- ANY vs IN operator
    - ANY can be used in sub-queries only, while IN can be used with/without sub-queries.
    - ANY can be used for comparision (<, >, <=, >=, =, or !=), while IN can be used only for equality comparision (=).
    - Both operators are logically similar to OR operator.
- ANY vs ALL operator
    - Both can be used for comparision (<, >, <=, >=, =, or !=).
    - Both are usable only with sub-queries.
    - ANY is similar to logical OR, while ALL is similar to logical AND.

```
-- display depts which do not have any emp.

SELECT * FROM dept WHERE deptno NOT IN (SELECT deptno FROM emp);
-- deptno != 10 AND deptno != 20 AND deptno != 30
-- NOT (deptno = 10 OR deptno = 20 OR deptno = 30)

SELECT * FROM dept WHERE deptno != ALL(SELECT deptno FROM emp);
```

Corelated sub-queries

- SELECT ... FROM table WHERE col = (SELECT ...)
- By default inner query is executed for each row of the outer query.
- If no optimization settings are enabled, sub-queries are slower than joins.

```
SELECT * FROM dept WHERE deptno IN (SELECT deptno FROM emp);
-- 10, ACC --> SELECT deptno FROM emp
-- 20, RES --> SELECT deptno FROM emp
-- 30, SAL --> SELECT deptno FROM emp
-- 40, OPS --> SELECT deptno FROM emp
```

- The sub-query execution can be speed-up if inner queries return/process less number rows.

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

- This is typically done by using WHERE clause in inner query.
- The WHERE clause in inner query may depend on current row of the outer query. This kind of query is called as "co-related sub-query".

```
SELECT * FROM dept d WHERE d.deptno IN
(SELECT e.deptno FROM emp e WHERE e.deptno = d.deptno);
-- 10, ACC --> SELECT e.deptno FROM emp e WHERE e.deptno = d.deptno --> 3 rows
(10,10,10)
-- 20, RES --> SELECT e.deptno FROM emp e WHERE e.deptno = d.deptno --> 5 rows
(20,20,20,20,20)
-- 30, SAL --> SELECT e.deptno FROM emp e WHERE e.deptno = d.deptno --> 6 rows
(30,30,30,30,30,30)
-- 40, OPS --> SELECT e.deptno FROM emp e WHERE e.deptno = d.deptno --> 0 rows
```

```
SELECT * FROM dept d WHERE EXISTS
(SELECT e.deptno FROM emp e WHERE e.deptno = d.deptno);
-- EXISTS check if sub-query return row(s).
```

```
-- display all depts which do not contain any emp.
SELECT * FROM dept d WHERE NOT EXISTS
(SELECT e.deptno FROM emp e WHERE e.deptno = d.deptno);
```

Sub-query in projection

```
-- display number of emps in each dept along with total number of emps.
SELECT deptno, COUNT(empno) FROM emp
GROUP BY deptno;

SELECT COUNT(empno) FROM emp;

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

-- +--------+--------------+
-- | deptno | COUNT(empno) |
-- +--------+--------------+
-- |     20 |            5 |
```

```
-- |     30 |            6 |
-- |     10 |            3 |
-- |   NULL |           14 |
-- +--------+--------------+

SELECT deptno,
COUNT(empno) AS deptcnt,
(SELECT COUNT(empno) FROM emp) AS totalcnt
FROM emp
GROUP BY deptno;
-- +--------+---------+----------+
-- | deptno | deptcnt | totalcnt |
-- +--------+---------+----------+
-- |     20 |       5 |       14 |
-- |     30 |       6 |       14 |
-- |     10 |       3 |       14 |
-- +--------+---------+----------+
```

Sub-query in FROM clause

- Inner-query can be written in FROM clause of SELECT statement. The output of the inner query is treated as a table (MUST give table alias) and outer query execute on that table.
- This is called as "Derived table" or "Inline view".

```
-- display empno, ename, sal and category of emp.
-- < 1500 -> POOR, 1500-2500 -> MID, > 2500 -> RICH
SELECT empno, ename, sal, CASE
WHEN sal < 1500 THEN 'POOR'
WHEN sal BETWEEN 1500 AND 2500 THEN 'MIDDLE'
ELSE 'RICH'
END AS category
FROM emp;
```

```
-- display number of emps in each category.
SELECT category, COUNT(empno) FROM
(SELECT empno, ename, sal, CASE
WHEN sal < 1500 THEN 'POOR'
WHEN sal BETWEEN 1500 AND 2500 THEN 'MIDDLE'
ELSE 'RICH'
END AS category
FROM emp) AS emp_category
GROUP BY category;
```

Sub-query in DML

```
-- INSERT new emp with name 'JOHN' and sal=2000 in dept 'OPERATIONS'.
SELECT deptno FROM dept WHERE dname = 'OPERATIONS';

INSERT INTO emp(ename, sal, deptno) VALUES ('JOHN', 2000, (SELECT deptno FROM dept
WHERE dname = 'OPERATIONS'));

SELECT * FROM emp;
```

```
-- give comm 100 to all emps in OPERATIONS dept.
UPDATE emp SET comm = 100 WHERE deptno =
(SELECT deptno FROM dept WHERE dname = 'OPERATIONS');

-- UPDATE emp SET comm = 100 WHERE deptno = 40;

SELECT * FROM emp;
```

```
-- delete emps from OPERATIONS dept.
DELETE FROM emp WHERE deptno =
(SELECT deptno FROM dept WHERE dname = 'OPERATIONS');

-- DELETE FROM emp WHERE deptno = 40;
```

- In MySQL, DML cannot be performed on the table from which inner query is selecting.

```
INSERT INTO emp(empno,ename,sal) VALUES(1000, 'JACK', 6000);

-- delete emp with max sal.
DELETE FROM emp
WHERE sal = (SELECT MAX(sal) FROM emp);
-- error: not allowed in MySQL

SET @maxsal = (SELECT MAX(sal) FROM emp);
DELETE FROM emp WHERE sal = @maxsal;
```

SQL Performance

- Modern RDBMS implement lot of optimization mechanisms (internally based on data structures and algorithms) like caching (materialization), semijoins or hashjoins, etc.
- These optimization logic will differ from RDBMS to RDBMS.

```
SELECT @@optimizer_switch;
```

- In MySQL, query Performance is measured in terms of query cost. Lower the cost, better is performance.
- The cost of query depends on data in the table(s), MySQL version, server machine config, optimizer settings, etc.

```
EXPLAIN FORMAT=JSON
SELECT * FROM emp WHERE sal < (SELECT MAX(sal) FROM emp WHERE deptno = 20);
-- 1.65

EXPLAIN FORMAT=JSON
SELECT * FROM emp WHERE sal < ANY(SELECT sal FROM emp WHERE deptno = 20);
-- 1.65
```

Views

- View is projection of the data.
- CREATE VIEW viewname AS SELECT ...;
- In MySQL views are non-materialized i.e. output of SELECT statement (of view) is not saved in server disk. Only SELECT query is saved in server disk.

```
SELECT empno, ename, sal, CASE
WHEN sal < 1500 THEN 'POOR'
WHEN sal BETWEEN 1500 AND 2500 THEN 'MIDDLE'
ELSE 'RICH'
END AS category
FROM emp;

CREATE VIEW v_empcategory AS
SELECT empno, ename, sal, CASE
WHEN sal < 1500 THEN 'POOR'
WHEN sal BETWEEN 1500 AND 2500 THEN 'MIDDLE'
ELSE 'RICH'
END AS category
FROM emp;

SHOW TABLES;

SHOW FULL TABLES;

SELECT * FROM v_empcategory;

SELECT category, COUNT(empno) FROM v_empcategory
GROUP BY category;

EXPLAIN FORMAT=JSON
SELECT category, COUNT(empno) FROM v_empcategory
GROUP BY category;
```

```
SHOW CREATE VIEW v_empcategory;

SELECT * FROM v_empcategory;

ALTER VIEW v_empcategory AS
SELECT empno, sal, CASE
WHEN sal < 1500 THEN 'POOR'
WHEN sal BETWEEN 1500 AND 2500 THEN 'MIDDLE'
ELSE 'RICH'
END AS category
FROM emp;

SELECT * FROM v_empcategory;

DROP VIEW v_empcategory;

SHOW FULL TABLES;
```

```
CREATE VIEW v_empsal AS
SELECT empno, ename, sal FROM emp;

SELECT * FROM v_empsal;

CREATE VIEW v_richemp AS
SELECT * FROM emp WHERE sal > 2500;

SELECT * FROM v_richemp;

CREATE VIEW v_empincome AS
SELECT empno, ename, sal, comm, sal + IFNULL(comm,0.0) income FROM emp;

SELECT * FROM v_empincome;

CREATE VIEW v_empjobsummary AS
SELECT job, SUM(sal) salsum, AVG(sal) salavg, MAX(sal) salmax, MIN(sal) salmin
FROM emp
GROUP BY job;

SELECT * FROM v_empjobsummary;

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

SELECT * FROM v_empjobsummary;
```

```
SELECT * FROM v_empsal;

INSERT INTO v_empsal VALUES (1001, 'JOY', 2300);
```

```
-- DML ops are allowed on Simple Views.

SELECT * FROM emp;
```

```
SELECT * FROM v_empjobsummary;

DELETE FROM v_empjobsummary WHERE job = 'DEV';
-- DML ops are NOT allowed on Complex Views.
```

MySQL - RDBMS

Agenda

- Views
- Security
- Transactions
- Row locking
- Indexes

Views

- Projection of data -- SELECT.
- CREATE VIEW viewname AS SELECT ...
- MySQL views are non-materialized.
- When query is executed on view, data is read from target table.
- Simple view: DQL + DML
- Complex view: DQL

```
SELECT USER(), DATABASE();
-- | sunbeam@localhost | classwork  |

SHOW FULL TABLES;

SHOW CREATE VIEW v_richemp;

SELECT empno, ename, sal FROM v_richemp;

INSERT INTO v_richemp(empno, ename, sal) VALUES(1000, 'JAMES', 2600);

SELECT empno, ename, sal FROM v_richemp;

INSERT INTO v_richemp(empno, ename, sal) VALUES(1001, 'HARRY', 2200);

SELECT empno, ename, sal FROM v_richemp;

SELECT empno, ename, sal FROM emp;

EXPLAIN FORMAT=JSON
SELECT empno, ename, sal FROM v_richemp;

INSERT INTO v_richemp(empno, ename, sal) VALUES(1002, 'MERRY', 2000);

ALTER VIEW v_richemp AS
SELECT * FROM emp WHERE sal > 2500
WITH CHECK OPTION;

INSERT INTO v_richemp(empno, ename, sal) VALUES(1003, 'ADAM', 1500);
-- error: CHECK OPTION failed -- sal <= 2500.
```

```
INSERT INTO v_richemp(empno, ename, sal) VALUES(1004, 'EVE', 2800);
```

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

SELECT * FROM v_richemp2;

SHOW CREATE VIEW v_richemp2;

DROP VIEW v_richemp;

SELECT * FROM v_richemp2;
-- error: invalid table/view.

DROP VIEW v_richemp2;
```

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

CREATE VIEW v_empdept AS
SELECT e.empno, e.ename, d.deptno, d.dname FROM emp e INNER JOIN dept d ON
e.deptno = d.deptno;

SELECT * FROM v_empdept;

-- display all emps in ACCOUNTING dept.
SELECT e.empno, e.ename, 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';

EXPLAIN FORMAT=JSON
SELECT e.empno, e.ename, d.deptno, d.dname FROM emp e INNER JOIN dept d ON
e.deptno = d.deptno
WHERE d.dname = 'ACCOUNTING';

EXPLAIN FORMAT=JSON
SELECT * FROM v_empdept
WHERE dname = 'ACCOUNTING';
```

- Assign: sales database
  - create view on all three tables joined together with all columns.
  - solve join assignments on that view.

JSON = Java Script Object Notation

- Data format to represent the data
- Another data format example is XML, CSV, ...
- JSON --> C struct like -- key-value pairs

```
"table": {
    "table_name": "d",
    "access_type": "ALL",
    "rows_examined_per_scan": 4,
    "rows_produced_per_join": 1,
}
```

- { ... } --> object
- [...] --> array
- "..." --> string
- 123.34, true, null --> number, bool

DCL

```
sunbeam> SHOW DATABASES;
-- classwork, sales, hr

root> SHOW GRANTS FOR sunbeam@localhost;

root> REVOKE ALL PRIVILEGES ON hr.* FROM sunbeam@localhost;

sunbeam> SHOW DATABASES;
-- classwork, sales
```

```
root> CREATE USER 'mgr'@'%' IDENTIFIED BY 'mgr';

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

root> CREATE USER 'dev1'@'localhost' IDENTIFIED BY 'dev1';

root> CREATE USER 'dev2'@'localhost' IDENTIFIED BY 'dev2';

root> SELECT user, host FROM mysql.user;

root> SHOW GRANTS FOR 'mgr'@'%';
-- USAGE -- no permissions

mgr> SHOW DATABASES;

root> GRANT ALL ON classwork.* TO 'mgr'@'%' WITH GRANT OPTION;

mgr> SHOW DATABASES;
```

```
mgr> GRANT INSERT, UPDATE, DELETE, SELECT ON classwork.emp TO
'teamlead'@'localhost';

mgr> GRANT INSERT, UPDATE, DELETE, SELECT ON classwork.dept TO
'teamlead'@'localhost';

mgr> GRANT INSERT, SELECT ON classwork.books TO 'teamlead'@'localhost';

root> SHOW GRANTS FOR 'teamlead'@'localhost';
```

```
teamlead> SHOW DATABASES;

teamlead> USE classwork;

teamlead> SHOW TABLES;

teamlead> DELETE FROM books;
-- error: Access denied

teamlead> INSERT INTO books VALUES (1, 'Atlas Shrugged', 'Ayn Rand', 'Novell',
727.29);

teamlead> SELECT * FROM books;
```

```
mgr> USE classwork;

mgr> CREATE VIEW v_empsummary AS
SELECT job, SUM(sal) salsum, AVG(sal) salavg FROM emp GROUP BY job;

mgr> GRANT SELECT ON classwork.v_empsummary TO 'dev1'@'localhost';

mgr> CREATE VIEW v_deptsummary AS
SELECT d.dname, SUM(e.sal) salsum, AVG(e.sal) salavg FROM emp e RIGHT JOIN dept d
ON e.deptno = d.deptno GROUP BY d.dname;

mgr> GRANT SELECT ON classwork.v_deptsummary TO 'dev1'@'localhost';

dev1> SHOW DATABASES;

dev1> USE classwork;

dev1> SHOW FULL TABLES;

dev1> SELECT * FROM v_deptsummary;

dev1> DESCRIBE v_deptsummary;
```

```
dev1> SHOW CREATE VIEW v_deptsummary;
-- error: Access denied.
```

```
root> SHOW GRANTS FOR 'teamlead'@'localhost';

root> REVOKE SELECT ON classwork.books FROM 'teamlead'@'localhost';

root> SHOW GRANTS FOR 'teamlead'@'localhost';

teamlead> SELECT * FROM books;
-- error: Access denied.
```

Transaction

```
-- sunbeam@localhost, classwork

CREATE TABLE accounts(id INT PRIMARY KEY, type CHAR(20),balance DECIMAL(9,2));

INSERT INTO accounts VALUES
(1, 'Saving', 20000.00),
(2, 'Current', 60000.00),
(3, 'Saving', 5000.00),
(4, 'Saving', 3000.00),
(5, 'Current', 50000.00),
(6, 'Saving', 10000.00);

SELECT * FROM accounts;

-- transfer Rs. 2000 from acc 1 to acc 3.
UPDATE accounts SET balance = balance - 2000
WHERE id = 1;

UPDATE accounts SET balance = balance + 2000
WHERE id = 3;

SELECT * FROM accounts;
```

- In MySQL, by default each DML operation is executed as a transaction with the single query and it is auto-committed.
- In MySQL, transaction is explicitly started using START TRANSACTION command. It is finalized using COMMIT command or disarded using ROLLBACK command.
- TCL commands
    - START TRANSACTION
    - COMMIT
    - ROLLBACK

```
-- transfer Rs. 5000 from acc 6 to acc 4.
START TRANSACTION;

UPDATE accounts SET balance = balance - 5000
WHERE id = 6;

SELECT * FROM accounts;

UPDATE accounts SET balance = balance + 5000
WHERE id = 4;

SELECT * FROM accounts;

COMMIT;
```

```
-- transfer Rs. 2000 from acc 6 to acc 4.
START TRANSACTION;

UPDATE accounts SET balance = balance - 2000
WHERE id = 6;

SELECT * FROM accounts;

UPDATE accounts SET balance = balance + 2000
WHERE id = 4;

SELECT * FROM accounts;

ROLLBACK;

SELECT * FROM accounts;
```

```
START TRANSACTION;

DELETE FROM accounts;

SELECT * FROM accounts;

ROLLBACK;

SELECT * FROM accounts;
```

```
DELETE FROM accounts;
-- delete without transaction
-- single dml query tx --> auto-committed.
-- START TX ++ DELETE ++ COMMIT.
```

```
-- all rows are deleted permanently.

ROLLBACK;
-- useless -- no current tx.

SELECT * FROM accounts;
```

- Transaction

```
START TRANSACTION;

dml1;
dml2;
dml3;

COMMIT;
-- changes of dml 1,2,3 are permanent.
-- transaction is completed.
```

```
START TRANSACTION;

dml1;
dml2;
dml3;

ROLLBACK;
-- changes of dml 1,2,3 are discarded.
-- transaction is completed.
```

- In MySQL, all DML operations are auto-commited (for each query).
- However MySQL can be configured to start transaction automatically after commit/rollback of previous transaction.

```
SELECT @@autocommit;

SET @@autocommit=0;

SELECT @@autocommit;

SELECT * FROM books;

DELETE FROM books;

ROLLBACK;

SELECT * FROM books;
```

```
SELECT * FROM dept;

DELETE FROM dept;

SELECT * FROM dept;

ROLLBACK;

SELECT * FROM dept;

SELECT * FROM accounts;

INSERT INTO accounts VALUES
(1, 'Saving', 20000.00),
(2, 'Current', 60000.00),
(3, 'Saving', 5000.00),
(4, 'Saving', 3000.00),
(5, 'Current', 50000.00),
(6, 'Saving', 10000.00);

SELECT * FROM accounts;

COMMIT;

SET @@autocommit=1;

SELECT @@autocommit;
```

- Transaction is set of DML queries executed as a single unit.
- Transaction is limited to same RDBMS server.

```
START TRANSACTION;

SELECT * FROM emp;

DELETE FROM emp WHERE empno < 1100;
-- changed in tx -- not permanent

SELECT * FROM emp;

TRUNCATE TABLE books;
-- ddl command -- current tx is auto committed.
-- DELETE FROM emp ... changes are permanent

SELECT * FROM books;

ROLLBACK;
-- has no effect ... tx is alreay commited.

SELECT * FROM emp;
```

```
START TRANSACTION;

SELECT * FROM dept;

DELETE FROM dept;

SELECT * FROM dept;

EXIT;
-- auto rollback current tx.
```

```
SELECT * FROM dept;
```

MySQL - RDBMS

Agenda

- Transactions
- Locking
- Indexes
- Constraints
- ALTER TABLE

Transactions

- In applications transactions are done programmatically.
    - Example 1: Funds transfer
        - accounts
    - Example 2: Online Ordering System
        - orders
        - order_items
        - payments
- JDBC -- Java database connectivity

```
try {
    // create connection
    con = DriverManager.getConnection(...);
    // start transaction
    con.setAutocommit(false);
    // create statements
    stmt1 = con.prepareStatement("INSERT ...");
    stmt1.executeUpdate();
    stmt2 = con.prepareStatement("INSERT ...");
    stmt2.executeUpdate();
    stmt3 = con.prepareStatement("INSERT ...");
    stmt3.executeUpdate();
    // commit transaction
    con.commit();
}
catch(Exception e) {
    // rollback transaction
    con.rollback();
}
```

- MySQL prompt

```
START TRANSACTION;
```

```
INSERT INTO orders VALUES(...);

INSERT INTO order_items VALUES(...);
INSERT INTO order_items VALUES(...);
INSERT INTO order_items VALUES(...);

INSERT INTO payments VALUES(...);

COMMIT; -- OR ROLLBACK;
```

Savepoints

- Savepoint is state of database within a transaction.
- User can rollback to any of the savepoint using
    - ROLLBACK TO sa;
- In this case all changes after savepoint "sa" are discarded.
- ROLLBACK TO "sa" do not ROLLBACK the whole transaction. The same transaction can be continued further (can make more DML queries).
- Transaction is completed only when COMMIT or ROLLBACK is done. All savepoint memory is released.
- COMMIT is not allowed upto a savepoint. We can only commit whole transaction.

```
START TRANSACTION;

INSERT INTO orders VALUES(...);
SAVEPOINT sa1;

INSERT INTO order_items VALUES(...);
INSERT INTO order_items VALUES(...);
INSERT INTO order_items VALUES(...);
SAVEPOINT sa2;

INSERT INTO payments VALUES(...);

ROLLBACK TO sa2;
-- revert db state back to sa2.
-- only payments query will be rollbacked.
-- transaction is not yet completed

INSERT INTO payments VALUES(....);
-- continue ops in same transaction

COMMIT; -- or ROLLBACK
-- transaction is completed.
```

```
START TRANSACTION;

INSERT INTO orders VALUES(...);
SAVEPOINT sa1;
```

```
INSERT INTO order_items VALUES(...);
INSERT INTO order_items VALUES(...);
INSERT INTO order_items VALUES(...);
SAVEPOINT sa2;

INSERT INTO payments VALUES(...);

ROLLBACK TO sa1;
-- revert db state back to sa1.
-- order_items & payment queries are rollbacked.

INSERT INTO order_items VALUES(...);
INSERT INTO payments VALUES(...);

COMMIT; -- or ROLLBACK
```

Transaction properties/characteristics

- Atomicity: All DML queries in transaction will be successful or failed/discarded. Partial transaction never committed.
- Consistent: At the end of transaction same state is visible to all the users.
- Isolation: Each transaction is isolated from each other. All transactions are added in a transaction queue at server side and process sequentially.
- Durability: At the end of transaction, final state is always saved (on server side).

Transaction Internals

```
root> SELECT * FROM accounts;

sunbeam> SELECT * FROM accounts;

sunbeam> START TRANSACTION;

sunbeam> DELETE FROM accounts WHERE id = 6;

sunbeam> SELECT * FROM accounts;
-- changes visible in current transaction

root> SELECT * FROM accounts;
-- changes not visible in other transactions

sunbeam> COMMIT;

sunbeam> SELECT * FROM accounts;

root> SELECT * FROM accounts;
-- changes are visible to other users after commit is done
```

```
sunbeam> START TRANSACTION;

sunbeam> DELETE FROM accounts WHERE id = 5;

sunbeam> SELECT * FROM accounts;

root> SELECT * FROM accounts;

sunbeam> ROLLBACK;

sunbeam> SELECT * FROM accounts;

root> SELECT * FROM accounts;
```

```
sunbeam> START TRANSACTION;

sunbeam> DELETE FROM accounts WHERE id = 5;

sunbeam> SELECT * FROM accounts;

root> SELECT * FROM accounts;

root> DELETE FROM accounts WHERE id = 4;
-- single dml tx -- autocommitted

root> SELECT * FROM accounts;

sunbeam> SELECT * FROM accounts;

sunbeam> COMMIT;

sunbeam> SELECT * FROM accounts;

root> SELECT * FROM accounts;
```

- When an user is in a transation, changes done by the user are saved in a temp table. These changes are visible to that user.
- However this temp table is not accessible/visible to other users and hence changes under progress in a transaction are not visible to other users.
- When an user is in a transaction, changes committed by other users are not visible to him. Because he is dealing with temp data.

Row locking

```
sunbeam> SELECT * FROM accounts;

sunbeam> START TRANSACTION;
```

```
sunbeam> DELETE FROM accounts WHERE id = 1;
-- row locked

sunbeam> SELECT * FROM accounts;

root> SELECT * FROM accounts;

root> UPDATE accounts SET balance = 10000 WHERE id = 1;
-- blocked

sunbeam> COMMIT;
-- root user unblocked

root> SELECT * FROM accounts;
```

```
sunbeam> SELECT * FROM accounts;

sunbeam> START TRANSACTION;

sunbeam> DELETE FROM accounts WHERE id = 2;
-- row locked

root> UPDATE accounts SET balance = 40000 WHERE id = 2;
-- blocked

sunbeam> ROLLBACK;
-- root user unblocked

root> SELECT * FROM accounts;

sunbeam> SELECT * FROM accounts;
```

```
sunbeam> SELECT * FROM accounts;

sunbeam> START TRANSACTION;

sunbeam> DELETE FROM accounts WHERE id = 2;
-- row locked

root> UPDATE accounts SET balance = 30000 WHERE id = 2;
-- root user is blocked
-- auto unblocked after some time if other user has not done COMMIT/ROLLBACK.

sunbeam> ROLLBACK;

sunbeam> SELECT * FROM accounts;
```

Pessimistic Locking

```
sunbeam> SELECT * FROM accounts;

sunbeam> START TRANSACTION;

sunbeam> SELECT * FROM accounts WHERE id = 2 FOR UPDATE;
-- row locked

root> SELECT * FROM accounts;

root> SELECT * FROM accounts WHERE id = 2 FOR UPDATE;
-- blocked

sunbeam> DELETE FROM accounts WHERE id = 2;

sunbeam> ROLLBACK;
-- root user unblocked
```

Table locking

```
sunbeam> SELECT * FROM depts;

sunbeam> START TRANSACTION;

sunbeam> DELETE FROM depts WHERE deptno = 40;
-- whole table is locked (bcoz no primary key)

root> DELETE FROM depts WHERE deptno = 30;
-- blocked

sunbeam> COMMIT;
-- root user unblocked

sunbeam> SELECT * FROM depts;

root> SELECT * FROM depts;
```

```
DESCRIBE accounts;

DESCRIBE depts;
```

Indexes

- Faster searching

Simple Index

```
SELECT * FROM books;

EXPLAIN FORMAT=JSON
SELECT * FROM books WHERE subject = 'C Programming';
-- 1.55

CREATE INDEX idx_books_subject ON books(subject);

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

DESCRIBE books;

SHOW INDEXES FROM books;

CREATE INDEX idx_books_author ON books(author DESC);

DESCRIBE books;

SHOW INDEXES FROM books;
```

```
EXPLAIN FORMAT=JSON
SELECT e.ename, d.dname FROM emps e
INNER JOIN depts d ON e.deptno = d.deptno;
-- 1.70

CREATE INDEX idx1 ON emps(deptno);
CREATE INDEX idx2 ON depts(deptno);

EXPLAIN FORMAT=JSON
SELECT e.ename, d.dname FROM emps e
INNER JOIN depts d ON e.deptno = d.deptno;
-- 1.62
```

Unique Index

- Duplicate values are not allowed.

```
CREATE UNIQUE INDEX idx3 ON emps(ename);

DESCRIBE emps;

SELECT * FROM emps;

SELECT * FROM emps WHERE ename = 'Nitin';
```

```
INSERT INTO emps VALUES (6, 'Rahul', 70, 1);
-- error: Duplicate entry

INSERT INTO emps VALUES (7, NULL, 50, 5);
-- (multiple) NULL value is allowed, but duplicate is not allowed.

CREATE UNIQUE INDEX idx4 ON emps(mgr);
-- error
```

Composite Index

```
SELECT * FROM emp;

DESCRIBE emp;

EXPLAIN FORMAT=JSON
SELECT * FROM emp WHERE deptno = 20 AND job = 'ANALYST';
-- 1.65

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

DESCRIBE emp;

SHOW INDEXES FROM emp;

EXPLAIN FORMAT=JSON
SELECT * FROM emp WHERE deptno = 20 AND job = 'ANALYST';
-- 0.70

EXPLAIN FORMAT=JSON
SELECT * FROM emp WHERE sal = 5000;

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

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

```
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 ON students(std,roll);

INSERT INTO students VALUES (1, 2, 'Ram', 99);
-- error: duplicate combination of std+roll not allowed

SELECT * FROM students;

INSERT INTO students VALUES (3, 1, 'Ram', 99);

DESCRIBE students;
```

Clustered Index

- Clustered index is auto-created on Primary key.
- It is internally a unique index that is used to lookup rows quickly on server disk.
- If Primary key is not present in the table, then a hidden (synthetic) column is created by RDBMS and Clustered index is created on it.

Drop Index

```
SHOW INDEXES FROM books;

DROP INDEX idx_books_author ON books;

DESCRIBE books;
```

Constraints

- Five Constraints
    - NOT NULL
    - Unique
    - Primary key
    - Foreign key
    - Check
- Types of constraints (based on syntax)
    - Column level

```
CREATE TABLE customers(
    id INT PRIMARY KEY,
    name CHAR(30) NOT NULL,
    email CHAR(40) UNIQUE NOT NULL,
    mobile CHAR(12) UNIQUE,
    addr VARCHAR(100)
);
```

        - NOT NULL, UNIQUE, PRIMARY, Foreign, CHECK
    - Table level

```
CREATE TABLE customers(
    id INT,
    name CHAR(30) NOT NULL,
    email CHAR(40) NOT NULL,
    mobile CHAR(12),
    addr VARCHAR(100),
    PRIMARY KEY(id),
    UNIQUE(email),
    UNIQUE(mobile)
);
```

        - UNIQUE, PRIMARY, Foreign, CHECK

NOT NULL

- NULL value is not allowed in the column.
- Can be given at column level only.

```
CREATE TABLE temp1(c1 INT, c2 INT, c3 INT NOT NULL);

DESCRIBE temp1;

INSERT INTO temp1 VALUES (1, 1, 1);
INSERT INTO temp1 VALUES (NULL, 2, 2);
INSERT INTO temp1(c1,c3) VALUES (3,3);
SELECT * FROM temp1;

INSERT INTO temp1 VALUES (4, 4, NULL);
-- error: c3 cannot be NULL
INSERT INTO temp1(c1,c2) VALUES (5,5);
-- error: c3 cannot be NULL

SHOW INDEXES FROM temp1;
```

Unique

- Cannot have duplicate value in the column.
- However NULL value(s) allowed.
- Unique constraint internally creates unique index.
- Unique constraint on combination of multiple columns internally creates Composite Unique index. Must be at table level -- UNIQUE(c1,c2).

```
CREATE TABLE temp2(c1 INT, c2 INT, UNIQUE(c1));
-- CREATE TABLE temp2(c1 INT UNIQUE, c2 INT);

INSERT INTO temp2 VALUES (1, 1);
INSERT INTO temp2 VALUES (2, 2);
INSERT INTO temp2 VALUES (3, 3);
INSERT INTO temp2 VALUES (4, 2);

INSERT INTO temp2 VALUES (3, 5);
-- error: c1 cannot be duplicated

INSERT INTO temp2 VALUES (NULL, 5);
INSERT INTO temp2 VALUES (NULL, 6);

SELECT * FROM temp2;

SHOW INDEXES FROM temp2;
```

```
DROP TABLE IF EXISTS students;

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

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 VALUES (1, 2, 'Ram', 99);
-- error: duplicate combination of std+roll not allowed
```

Primary key

- Primary key --> Column(s)
- "Identity" of each row/record.
- Primary key is "like" Unique constraint (cannot be duplicated) + NOT NULL constraint (cannot be NULL).
- In a table there is single primary key, but a table can have multiple unique key (constraints).

```
CREATE TABLE cdac_students(
    prn CHAR(16) PRIMARY KEY,
    name CHAR(40) NOT NULL,
    email CHAR(30) UNIQUE NOT NULL,
    mobile CHAR(12) UNIQUE NOT NULL,
```

```
    addr VARCHAR(100)
);
```

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

```
DROP TABLE IF EXISTS students;

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

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 VALUES (1, 2, 'Ram', 99);
-- error: duplicate combination of std+roll not allowed

DESCRIBE students;
```

- The Primary key internally creates UNIQUE index by name "PRIMARY".

```
SHOW INDEXES FROM cdac_students;

SHOW INDEXES FROM students;
```

CREATE TABLE t (c1 INT DEFAULT 0, ...)

MySQL - RDBMS

Agenda

- Constraints
  - Surrogate Primary Key
  - Foreign Key
  - Check
- ALTER table
- PSM / PL-SQL
  - Stored procedure
  - ~~Functions~~
  - ~~Triggers~~

Constraints

- Restrict values in the column.
- Constraints are checked/verified while performing DML operations. It slow down DML operations.
- Constraints ensure valid data is entered in the database.
- Unique key, Primary key and Foreign key constraints internally create indexes. It will help searching faster on these keys/columns.

Primary Key

- Primary key --> Identity of row/record/tuple.
- Natural primary key

```
CREATE TABLE customers(
    email CHAR(40) PRIMARY KEY,
    password CHAR(40),
    name CHAR(40),
    addr CHAR(100),
    birth DATE
);
```

- Composite primary key

```
CREATE TABLE students(
    email CHAR(40),
    password CHAR(40),
    name CHAR(40),
    grade CHAR(2),
    course_code CHAR(20),
    PRIMARY KEY(course_code, email)
);
```

- Surrogate primary key
    - Usually auto-generated.
    - Oracle/Pg-SQL --> Sequences
    - MS-SQL --> Identity
    - MySQL --> AUTO_INCREMENT

```
CREATE TABLE students(
    regno INT AUTO_INCREMENT,
    email CHAR(40),
    password CHAR(40),
    name CHAR(40),
    grade CHAR(2),
    course_code CHAR(20),
    PRIMARY KEY(regno)
);
```

```
CREATE TABLE items(
    id INT PRIMARY KEY AUTO_INCREMENT,
    name CHAR(30),
    price DECIMAL(5,2)
);

INSERT INTO items(name,price) VALUES('A', 10);
INSERT INTO items(name,price) VALUES('B', 15);
INSERT INTO items(name,price) VALUES('C', 20);
SELECT * FROM items;

ALTER TABLE items AUTO_INCREMENT = 100;

INSERT INTO items(name,price) VALUES('X', 50);
INSERT INTO items(name,price) VALUES('Y', 55);
SELECT * FROM items;

INSERT INTO items(id,name,price) VALUES (1000, 'P', 30);
SELECT * FROM items;
INSERT INTO items(name,price) VALUES('Q', 60);
SELECT * FROM items;
```

Foreign Key

```
DESCRIBE emps;

DESCRIBE depts;

SELECT * FROM depts;
```

```
SELECT * FROM emps;

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');

DESCRIBE depts;

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

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

SELECT * FROM depts;
SELECT * FROM emps;

INSERT INTO emps VALUES (6, 'Vishal', NULL, 3);
-- FK can be NULL
SELECT * FROM emps;

SELECT * FROM depts;
DELETE FROM depts WHERE deptno=40;

DELETE FROM depts WHERE deptno=30;
-- error: a foreign key constraint fails

DROP TABLE depts;
-- error: Cannot drop table 'depts' referenced by a foreign key constraint
```

- depts "1" ---- "*" emps
  - Parent-child relationship
  - Parent = depts table
  - Child = emps table
- Foreign key

- Cannot add/update in child row, if corresponding row is absent in parent table.
- Cannot delete parent row, if corresponding rows are present in child table.

```
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');

DESCRIBE depts;

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);

SELECT * FROM depts;
SELECT * FROM emps;

DELETE FROM depts WHERE deptno = 20;
-- ON DELETE CASCADE: If parent row is deleted, corresponding child rows will be
deleted automatically.

SELECT * FROM depts;
SELECT * FROM emps;

UPDATE depts SET deptno=100 WHERE dname='DEV';
-- ON UPDATE CASCADE: If parent row (primary key) is updated, corresponding child
rows (foreign key) will be updated automatically.

SELECT * FROM depts;
SELECT * FROM emps;

DROP TABLE depts;
-- error: Cannot drop table 'depts' referenced by a foreign key constraint
```

- Foreign is mapped to the primary key of other table.
- If PK is Composite primary key, the Foreign key can be Composite key.

```
CREATE TABLE students(
    email CHAR(40),
    password CHAR(40),
    name CHAR(40),
    grade CHAR(2),
```

```
    course_code CHAR(20),
    PRIMARY KEY(course_code, email)
);

CREATE TABLE marks(
    id INT,
    subject CHAR(20),
    marks INT,
    course_id CHAR(20),
    email CHAR(40),
    FOREIGN KEY (course_id,email) REFERENCES students(course_code, email);
);
```

- Foreign key internally creates index on the table. It also helps in faster searching.

```
DESCRIBE emps;

SHOW INDEXES FROM emps;
```

- Foreign key constraint can be disabled temporarily in some cases (like backup/restore).

```
SELECT @@foreign_key_checks;

CREATE TABLE dept_backup(deptno INT, dname CHAR(40), loc CHAR(40), PRIMARY
KEY(deptno));

CREATE TABLE emp_backup(empno INT, ename CHAR(40), sal DECIMAL(8,2), deptno
INT,
PRIMARY KEY(empno), FOREIGN KEY (deptno) REFERENCES dept_backup(deptno));

INSERT INTO dept_backup SELECT * FROM dept;
SELECT * FROM dept_backup;

SET @@foreign_key_checks=0;

INSERT INTO emp_backup(empno,ename,sal,deptno) SELECT empno,ename,sal,deptno
FROM emp;
-- insert is fast, bcoz FK is disabled.

INSERT INTO emp_backup VALUES(1000, 'JOHN', 2000, 60);
-- allowed, bcoz FK checks are disabled -- but wrong

SET @@foreign_key_checks=1;
-- FK check is enabled -- further DML ops.

INSERT INTO emp_backup VALUES(1001, 'JACK', 2200, 60);
-- error: FK checks are enabled

SELECT * FROM emp_backup;
```

- Foreign key can be for the same table. It is called as "self-referencing" FK.

```
CREATE TABLE emps(
    empno INT,
    ename CHAR(40),
    mgr INT,
    PRIMARY KEY(empno),
    FOREIGN KEY(mgr) REFERENCES emps(empno)
);
```

Check

- Arbitrary conditions (application specific) to be applied on the column.
- Do not work in MySQL version <= 8.0.15

```
CREATE TABLE employees(
    id INT PRIMARY KEY,
    ename CHAR(40) CHECK (LENGTH(ename) > 1),
    age INT NOT NULL CHECK (age > 18),
    sal DECIMAL(7,2) CHECK (sal > 1000),
    comm DECIMAL(7,2),
    CHECK((sal + IFNULL(comm,0)) > 1200)
);
```

```
INSERT INTO employees VALUES (1, 'A', 20, 2000, NULL);
-- error: LENGTH(ename) > 1
INSERT INTO employees VALUES (1, 'Om', 16, 2000, NULL);
-- error: age > 18
INSERT INTO employees VALUES (1, 'Om', 20, 900, NULL);
-- error: sal > 1000
INSERT INTO employees VALUES (1, 'Om', 20, 1100, NULL);
-- error: (sal + IFNULL(comm,0)) > 1200
INSERT INTO employees VALUES (1, 'Om', 20, 1100, 200);
-- okay
```

Constraint names

```
CREATE TABLE employees(
    id INT,
    ename CHAR(40),
    age INT NOT NULL,
    sal DECIMAL(7,2),
    comm DECIMAL(7,2),
```

```
    deptno INT,
    PRIMARY KEY(id),
    FOREIGN KEY(deptno) REFERENCES departments(deptno),
    UNIQUE(ename),
    CHECK((sal + IFNULL(comm,0)) > 1200)
);
-- names of constraints are given auto by db
```

```
CREATE TABLE employees(
    id INT,
    ename CHAR(40),
    age INT NOT NULL,
    sal DECIMAL(7,2),
    comm DECIMAL(7,2),
    deptno INT,
    CONSTRAINT pk_employees PRIMARY KEY(id),
    CONSTRAINT fk_dept FOREIGN KEY(deptno) REFERENCES departments(deptno),
    CONSTRAINT uk_ename UNIQUE(ename),
    CONSTRAINT chk_income CHECK((sal + IFNULL(comm,0)) > 1200)
);
```

Show Constraints

```
SHOW CREATE TABLE emps;

SELECT TABLE_NAME,
       COLUMN_NAME,
       CONSTRAINT_NAME,
       REFERENCED_TABLE_NAME,
       REFERENCED_COLUMN_NAME
FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
WHERE TABLE_SCHEMA = 'classwork'
      AND TABLE_NAME = 'emps'
      AND REFERENCED_COLUMN_NAME IS NOT NULL;
```

ALTER Table

- CREATE TABLE -- Table Structure (Metadata)
- DML operations -- Table Data
- ALTER TABLE -- Change table structure/metadata
    - Add column, Remove column, Change column data type/name, Add/Remove constraint, ...
    - Not recommeded in production database.
    - After alteration table storage become unefficient.

```
DESCRIBE emp_backup;
```

```
ALTER TABLE emp_backup ADD COLUMN job CHAR(20);

SELECT * FROM emp_backup;

UPDATE emp_backup e SET e.job = (SELECT job FROM emp WHERE empno = e.empno);

SELECT * FROM emp_backup;

DESCRIBE emp_backup;

ALTER TABLE emp_backup MODIFY job VARCHAR(40);
-- can change data type to compatible data type

DESCRIBE emp_backup;

ALTER TABLE emp_backup MODIFY job INT;
-- error: cannot change data type to incompatible.

ALTER TABLE emp_backup CHANGE ename name CHAR(30);

DESCRIBE emp_backup;

ALTER TABLE emp_backup DROP COLUMN sal;

DESCRIBE emp_backup;
```

```
ALTER TABLE emp ADD PRIMARY KEY (empno);

ALTER TABLE emp ADD UNIQUE(ename);

SHOW CREATE TABLE emp;

ALTER TABLE dept ADD PRIMARY KEY (deptno);

ALTER TABLE emp ADD FOREIGN KEY (deptno) REFERENCES dept (deptno);
```

```
ALTER TABLE emp DROP PRIMARY KEY;

SHOW CREATE TABLE emp;

ALTER TABLE emp DROP CONSTRAINT ename;

ALTER TABLE emp DROP CONSTRAINT emp_ibfk_1;
```

PSM / PL-SQL

Stored procedure

- Default DELIMITER is semicolon.
- When ; is found, client submit the code/query to the server.
- It should be changed temporarily to implement stored procedure using DELIMITER keyword.

Steps of Stored Procedure programming.

- step 1: Create a .sql file (like psm01.sql).
- step 2: Use SOURCE command on mysql CLI to execute it.

```
SOURCE D:/sep21/DAC/dbt/day09/psm01.sql
```

- step 3: Call the procedure.

```
CALL sp_hello1();
```

Stored Procedure Result into Table

```
CREATE TABLE result(id INT, val CHAR(100));
```

Stored Procedure Params

```
// arg n --> input to function --> in param
int sqr(int n) {
    return n * n;
}
void main() {
    // ...
    res = sqr(5);
    // ...
}
```

```
// arg n --> input to function --> in param
// arg r --> output from function --> out param
void sqr(int n, int *r) {
    *r = n * n;
}
void main() {
    // ...
    sqr(5, &res);
    // ...
}
```

```c
// arg n --> input to fn & output from fn --> in-out param
void sqr(int *n) {
    *n = (*n) * (*n);
}
void main() {
    // ...
    res = 5;
    res = sqr(&res);
    // ...
}
```

```sql
CREATE PROCEDURE sp_sqr1(IN p_n INT, OUT p_r INT)
BEGIN
    SET p_r = p_n * p_n;
END
```

```sql
CALL sp_sqr1(5, @res1)
SELECT @res1;
```

```sql
CREATE PROCEDURE sp_sqr2(INOUT p_n INT)
BEGIN
    SET p_n = p_n * p_n;
END
```

```sql
SET @res2 = 5;
CALL sp_sqr2(@res2);
SELECT @res2;
```

MySQL - RDBMS

Agenda

- PSM / PL-SQL
  - Exception handling
  - Cursors
  - Functions
  - Triggers
- Normalization
  - SQL keys
  - UNF
  - 1-NF
  - 2-NF
  - 3-NF
  - BCNF
  - Denormalization

PSM

Stored Procedure

- Login with "root" and use "classwork" database.

```
SELECT USER(), DATABASE();
-- | root@localhost | classwork  |
SELECT DEFINER, ROUTINE_DEFINITION FROM INFORMATION_SCHEMA.ROUTINES
WHERE DEFINER = 'sunbeam@localhost';
```

Exception handling

- In MySQL errors are represented with error code or error state.
  - Error code
    - 1062 -- Duplicate entry
    - 1044 -- Access denied
    - 1146 -- Table doesn't exists
  - Error state
    - 23000 -- Duplicate entry
    - 42000 -- Access denied
    - 42S02 -- Table doesn't exists
    - NOT FOUND -- End of file/cursor

```
SELECT * FROM books;
```

```
INSERT INTO books VALUES (4003, 'Harry Potter', 'Rowling', 'Novell', 626.9);
-- ERROR 1062 (23000): Duplicate entry
```

- Transaction management using error handler

```
CREATE PROCEDURE sp_placeorder(...)
BEGIN
    DECLARE EXIT HANDLER FOR error
    BEGIN
        ROLLBACK;
        SELECT 'Order failed' AS msg;
    END;

    START TRANSACTION;
    INSERT INTO orders VALUES (...);
    INSERT INTO order_items VALUES (...);
    INSERT INTO payments VALUES (...);
    COMMIT;
END;
```

Cursors

- In Java, foreach loop is used to access elements one by one from a collection.

```
for(Emp e : emps) {
    // ...
}
```

- 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.

```
DECLARE v_flag INT DEFAULT 0;
DECLARE CONTINUE HANDLER FOR NOT FOUND -- 1
BEGIN
    SET v_flag = 1;
END;
```

```
DECLARE v_cur CURSOR FOR SELECT ...; -- 2

OPEN v_cur; -- 3

label: LOOP
    FETCH v_cur INTO variable(s); -- 4
    IF v_flag = 1 THEN  -- 5
        LEAVE label;    -- 6
    END IF;
    process variables; -- 4
END LOOP;

CLOSE v_cur; -- 6
```

Cursor - use case

- T1 (C1) --> 1, 2, 3, 4
- T2 (C2) --> 10, 22, 35, 46

```
DECLARE v_cur1 CURSOR FOR SELECT c1 FROM t1;
DECLARE v_cur2 CURSOR FOR SELECT c2 FROM t2;

OPEN v_cur1;
OPEN v_cur2;

SET v_i = 1;

again: LOOP
    FETCH v_cur1 INTO v1;
    IF v_flag = 1 THEN
        LEAVE again;
    END IF;

    FETCH v_cur2 INTO v2;
    IF v_flag = 1 THEN
        LEAVE again;
    END IF;

    INSERT INTO result VALUES (v_i, CONCAT(v1, ' - ', v2));
    SET v_i = v_i + 1;
END LOOP;

CLOSE v_cur1;
CLOSE v_cur2;
```

Characteristics of "MySQL Cursors"

- 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.
- 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.
- 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.

User-defined Functions

- 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.
- 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.

Triggers

- Stored Proceduer and Functions
    - PSM syntax
    - Stored on server disk
    - Processed on server side
    - Reusable
    - Called by user
        - CALL sp_name()
        - SELECT fn_name()
- Trigger is MySQL program (PSM syntax). It's execution is triggered (caused) by some event -- DML operation on a table.
    - BEFORE INSERT
    - AFTER INSERT
    - BEFORE UPDATE
    - AFTER UPDATE
    - BEFORE DELETE
    - AFTER DELETE
- If multiple rows are INSERT/UPDATE/DELETE, then trigger will be executed once "for each row".
- The affected rows can be accessed using NEW and OLD keywords.
    - INSERT --> NEW row

    - DELETE --> OLD row
    - UPDATE --> NEW and OLD row
- It is never called explicitly by the user. It cannot have arguments or return value. It's output is not printed console.

```
DROP TABLE IF EXISTS accounts;

CREATE TABLE accounts(id INT PRIMARY KEY, type CHAR(10), balance DECIMAL(9,2));
INSERT INTO accounts VALUES
(1, 'Saving', 0.0),
(2, 'Saving', 0.0),
(3, 'Saving', 0.0);

CREATE TABLE transactions(id INT PRIMARY KEY AUTO_INCREMENT, acc_id INT, type
CHAR(20), amount DECIMAL(9,2));
```

```
CREATE TRIGGER update_balance
AFTER INSERT ON transactions
FOR EACH ROW
BEGIN
    IF NEW.type = 'Deposit' THEN
        UPDATE accounts SET balance = balance + NEW.amount WHERE id = NEW.acc_id;
    ELSE
        UPDATE accounts SET balance = balance - NEW.amount WHERE id = NEW.acc_id;
    END IF;
END;
```

```
-- cascading trigger
CREATE TRIGGER balance_check
AFTER UPDATE ON accounts
FOR EACH ROW
BEGIN
    IF NEW.balance < 0 THEN
        -- error
    END IF;
END;
```

```
INSERT INTO transactions(acc_id, type, amount) VALUES (1, 'Deposit', 2000.0);
INSERT INTO transactions(acc_id, type, amount) VALUES (2, 'Deposit', 10000.0);
INSERT INTO transactions(acc_id, type, amount) VALUES (2, 'Withdraw', 1000.0);
```