PG-DAC February 2020

Introduction to DBMS

- Duration : 60 hours
  - Theory : 30 hours
  - Lab : 30 hours

- Evaluation
  - Theory Exam : 40% [CDAC is going to conduct]
  - Lab Exam : 40% [SunBeam is going to conduct]
  - Internal Exam : 20% [SunBeam is going to conduct]

- Database : MySQL

- Reference Book
  - MySQL Developer’s Library – Paul DuBois(Pearson)

Session 1

Introduction to DBMS

What is DBMS

Need of DBMS

Areas Where DBMS are used

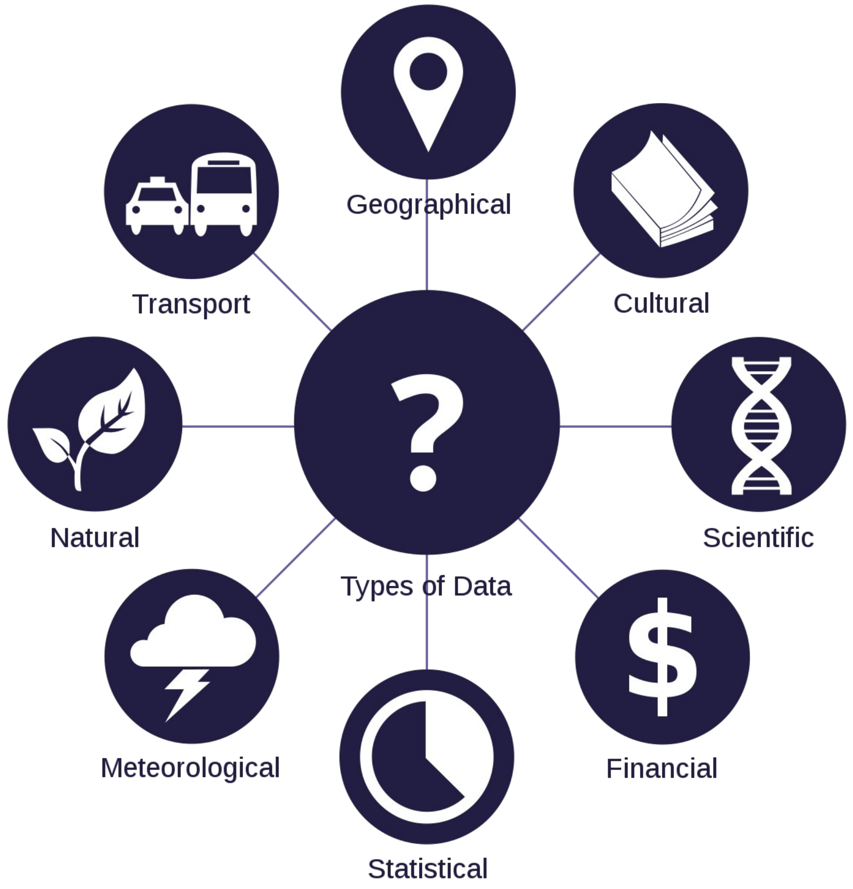
Data Models

Relational Data Model

What is *Data?*

- ➢ An unorganized and raw things.
- ➢ It is simple and useless until we process it.
- ➢ Example:
  - Random numbers
  - Unknown characters
  - Symbol
  - Images

What is *Data?*


Geographical
Transport
Cultural
Natural
Scientific
Types of Data
Meteorological
Financial
Statistical

What is *Information*?

- The work done by the computer to convert data into processing is called processing.
- The process of searching, sorting, grouping or calculating data is called data processing.
- Processed data is called as information.

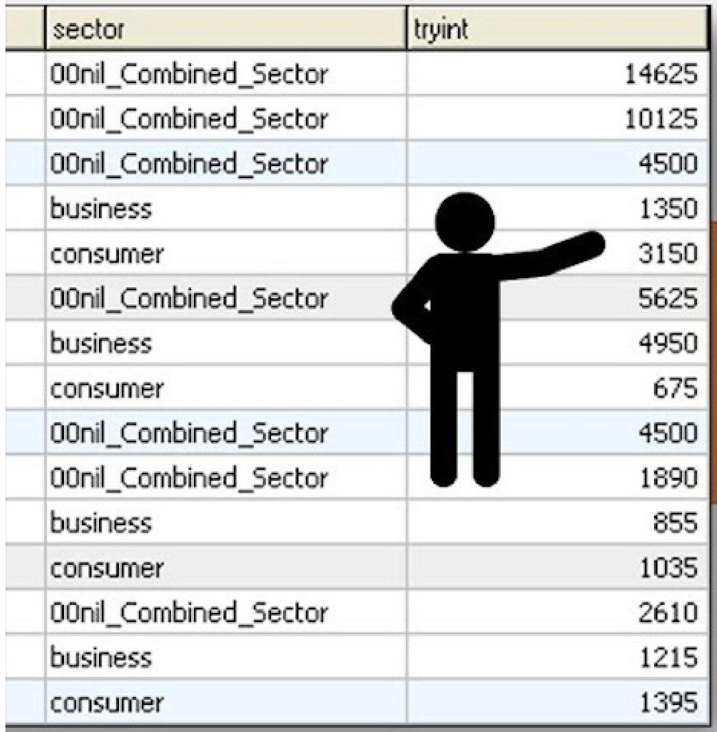
What is Information?

- What does the number 23071983 means?
  - A birthday?(23rd July 1983)
  - A bank account number?
  - A telephone number?
  - A club membership number?

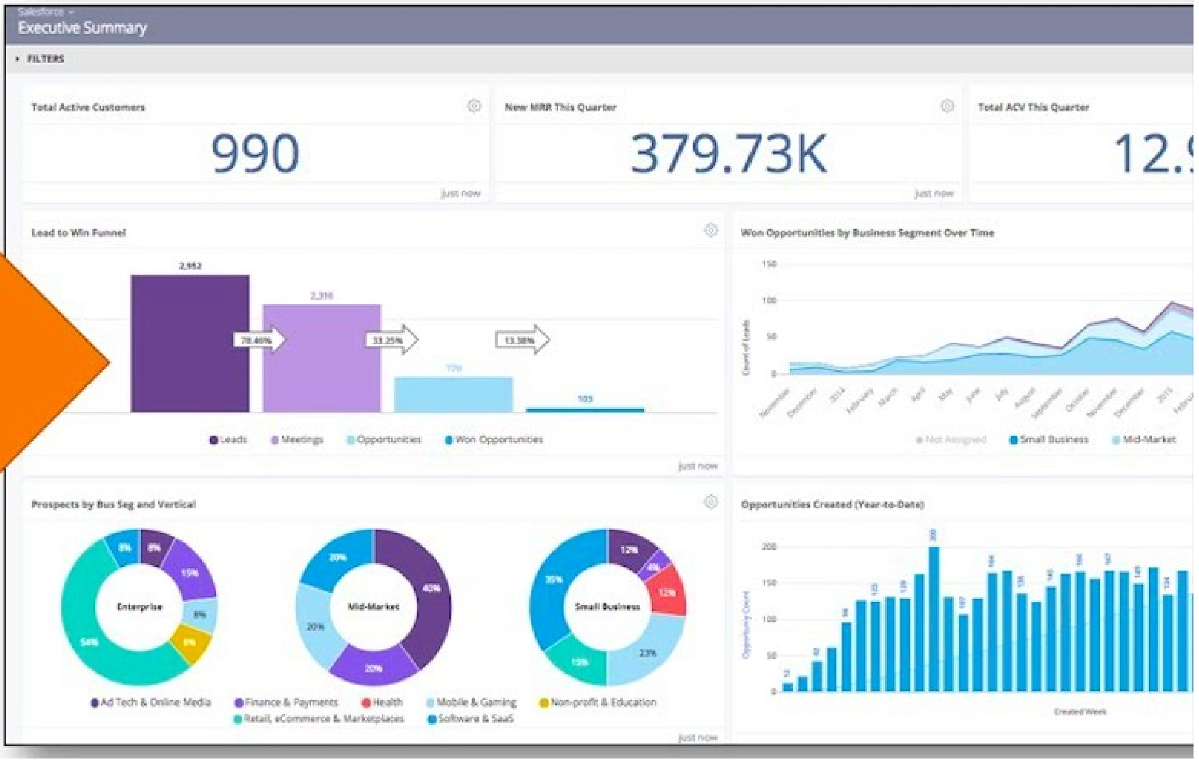
- Without processing data is meaningless.

Data

Information



	sector	tryint
	00nil_Combined_Sector	14625
	00nil_Combined_Sector	10125
	00nil_Combined_Sector	4500
	business	1350
	consumer	3150
	00nil_Combined_Sector	5625
	business	4950
	consumer	675
	00nil_Combined_Sector	4500
	00nil_Combined_Sector	1890
	business	855
	consumer	1035
	00nil_Combined_Sector	2610
	business	1215
	consumer	1395



What is Database?

- ➢It is a collection of large amount of data.
- ➢It contains coherent and meaningful data.
- ➢Common Examples
  - Address Book
  - Telephone Dictionary
  - Students Record

What is Database?

- Example: Postal Address.
  - A building name
  - A flat number
  - A road name
  - An area name
  - A state name
  - A pincode
  - A country name
- The postal address is data and Address book is a database.
- The address book could be termed as a coherent collection of data.

What is DBMS?

- *Database Management System*.
- It is a readymade software that helps to manage the data.
- ANSI definition is :

  It is collection of interrelated data(database) and a set of programs to access those data.

What is DBMS?

- The primary goal of DBMS is to provide a way to store and retrieve database information that is both convenient and efficient.
- It allows us to insert, update, delete and process data.
- Example:
  - MS Excel, FoxPro, Tally etc.

Need Of DBMS

In the early days, database applications were built directly on top of file systems, which leads to:

- *Data redundancy and inconsistency*
  - Data is stored in multiple file formats resulting in duplication of information in different files
- *Difficulty in accessing data*
  - Need to write a new program to carry out each new task
- *Data isolation*
  - Multiple files and formats

Need Of DBMS

- *Integrity Problems*
  - *Data integrity refers to the problem of ensuring that database contains only accurate data.*

- *Atomicity Problems*
  - Failures may leave database in an inconsistent state with partial updates carried out
  - Example: Transfer of funds from one account to another should either complete or not happen at all

Need Of DBMS

- Concurrent access by multiple users
  - Concurrent access needed for performance
  - Uncontrolled concurrent accesses can lead to inconsistencies
    - Ex: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time

- *Security problems*
  - Hard to provide user access to some, but not all, data.

Database management system offers solution to the all above problems.

Database System Applications

- Enterprise Information
  - Sales: For customer, product, and purchase information.
  - Accounting: For payments, receipts, account balances, assets and other accounting information.
  - Human resources: For information about employees, salaries, payroll taxes, and benefits, and for generation of pay checks.
  - Manufacturing: For management of the supply chain and for tracking production of items in factories, inventories of items in warehouses and stores, and orders for items.
  - Online retailers: For sales data noted above plus online order tracking, generation of recommendation lists, and maintenance of online product evaluations.

Database System Applications

- *Banking and Finance*
  - *Banking*: For customer information, accounts, loans, and banking transactions.
  - *Credit card transactions*: For purchases on credit cards and generation of monthly statements.
  - *Finance*: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds; also for storing real-time market data to enable online trading by customers and automated trading by the firm.
- Navigation systems:
  - For maintaining the locations of varies places of interest along with the exact routes of roads, train systems, buses, etc.

Database System Applications

- *Universities*:
  - For student information, course registrations, and grades.

- *Airlines*:
  - For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.

- *Telecommunication*:
  - For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks

Data Models

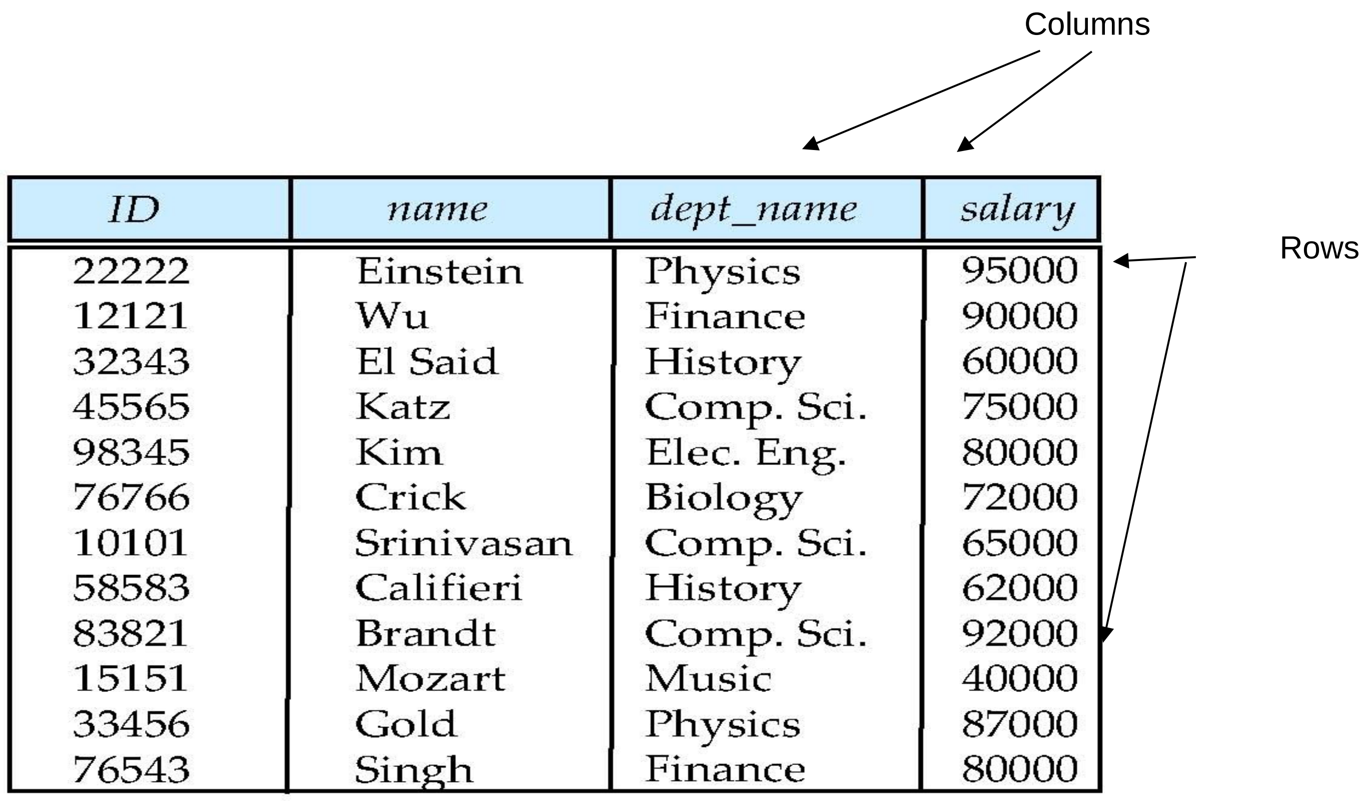
- A collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.
- A data model provides a way to describe the design of a database at the physical, logical, and view levels.
- There are number of different data models:
  1. Relational
  2. Entity Relationship (Mainly for DB design)
  3. Object-Based (Object-oriented and Object-relational)
  4. Semistructured (XML)
  5. Other older Models:
     1. Hierarchical Data Model (tree like structure)
     2. Network Data Model

Relational Data Model

- It uses a collection of Relations to represent both data and the relationships among those data. Relation is also called as ”**Table / Entity Class**”.
- Each table contains records of a particular type. Record is also called as “**Row or Tuple/Entity**”.
- Each record type defines a fixed number of fields, or attributes. Attribute is also called as “**Column**”.
- Collection of columns and rows is a table.

Relational Data Model

- The model was introduced by E.F. Codd in 1970.
- All the data is stored in tables.


Columns
Rows

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table

Data Types

- MySQL supports SQL data types in several categories:
  - Numeric types
  - Date and time types
  - String (character and byte) types
  - Spatial types
  - The JSON data type.

Conventions

```
1. For integer types, M indicates the maximum display width. As of MySQL
8.0.17, the display width attribute is deprecated for integer data types
and support for it will be removed in a future MySQL version.
2. For floating-point and fixed-point types, M is the total number of
digits that can be stored (the precision).
3. For string types, M is the maximum length.
4. D applies to floating-point and fixed-point types and indicates the
number of digits following the decimal point (the scale). The maximum
possible value is 30, but should be no greater than M−2.
5. fsp applies to the TIME, DATETIME, and TIMESTAMP types and represents
fractional seconds precision; that is, the number of digits following the
decimal point for fractional parts of seconds. The fsp value, if given,
must be in the range 0 to 6. A value of 0 signifies that there is no
fractional part. If omitted, the default precision is 0.
```

Numeric Data Types

- MySQL supports all standard SQL numeric data types. These types include the exact numeric data types as well as the approximate numeric data types.
- Exact numeric data types:
  1. SMALLINT
  2. INTEGER
  3. DECIMAL
  4. NUMERIC
- Approximate numeric data types
  1. FLOAT
  2. REAL
  3. DOUBLE PRECISION
- The keyword INT is a synonym for INTEGER.
- The keywords DEC and FIXED are synonyms for DECIMAL.
- MySQL treats REAL and DOUBLE are synonym for DOUBLE PRECISION
- The BIT data type stores bit values and is supported for MyISAM, MEMORY, InnoDB, and NDB tables.
- Following are the MySQL Numeric Data types
  1. BIT(M)

- The BIT data type is used to store bit values. A type of BIT(M) enables storage of M-bit values.
- M indicates the number of bits per value.
- It can be between 1 to 64. The default is 1 if M is omitted.
- Bit-value literals are written using b'val' or 0bval notation. val is a binary value written using zeros and ones.
    - b'2' => (2 is not a binary digit)
    - 0B01 => (0B must be written as 0b)
- By default, a bit-value literal is a binary string : For example, b'111' and b'10000000'

```
CREATE TABLE bit_test( bytes BIT(4));
INSERT INTO bit_test VALUES( b'10');    -- OK
INSERT INTO bit_test VALUES( b'100');   -- OK
INSERT INTO bit_test VALUES( b'1000');  -- OK
INSERT INTO bit_test VALUES( b'10001'); -- ERROR: Data too long
```

```
    * Bit values in result sets are returned as binary values, which may
not display well.
    * To convert a bit value to printable form, use a conversion function
such as BIN() or HEX().
```

```
select bytes + 0 from bit_test;
SELECT BIN(bytes) FROM bit_test;
SELECT OCT(bytes) FROM bit_test;
SELECT HEX(bytes) FROM bit_test;
```

```
    * Size of BIT is approximately (M+7)/8 bytes.
2. TINYINT
    * A very small integer.
    * The signed range is -128 to 127.
    * The unsigned range is 0 to 255.
    * Size of TINYINT in 1 byte.
```

```
CREATE TABLE tinyint_test( id TINYINT(2));
CREATE TABLE tinyint_test( id TINYINT UNSIGNED );
CREATE TABLE tinyint_test( id TINYINT ZEROFILL );
CREATE TABLE tinyint_test( id TINYINT(2) ZEROFILL );
```

```
    * When used in conjunction with the optional ZEROFILL attribute, the
default padding of spaces is replaced with zeros.
```

```
3. BOOL/BOOLEAN
     * These types are synonyms for TINYINT(1)
     * A value of zero is considered false. Nonzero values are considered
true
```

```
CREATE TABLE boolean_test( status BOOLEAN );
CREATE TABLE boolean_test( status BOOL );
CREATE TABLE boolean_test( status TINYINT(1) );
```

```
INSERT INTO status VALUES( true );
INSERT INTO boolean_test (status ) VALUES (true);
```

```
4. SMALLINT
     * A small integer.
     * The signed range is -32768 to 32767.
     * The unsigned range is 0 to 65535.
     * Its size is 2 bytes.
5. MEDIUMINT
     * A medium-sized integer.
     * The signed range is -8388608 to 8388607.
     * The unsigned range is 0 to 16777215.
     * Its size is 3 bytes.
6. INT( synonyms : INTEGER )
     * A normal-size integer.
     * The signed range is -2147483648 to 2147483647.
     * The unsigned range is 0 to 4294967295.
     * Its size is 4 bytes.
7. BIGINT
     * A large integer.
     * The signed range is -9223372036854775808 to
       9223372036854775807.
     * The unsigned range is 0 to 18446744073709551615.
     * It's size is 8 bytes.
8. DECIMAL
     * It is synonym for DEC, FIXED.
     * A packed "exact" fixed-point number.
     * In MySQL, NUMERIC is implemented as DECIMAL, so the following remarks
about DECIMAL apply equally to NUMERIC.
     * M is the total number of digits (the precision) and D is the number
of digits after the decimal point (the scale).
     * The decimal point and (for negative numbers) the - sign are not
counted in M.
     * If D is 0, values have no decimal point or fractional part.
     * The maximum number of digits (M) for DECIMAL is 65.
     * The maximum number of supported decimals (D) is 30.
     * If D is omitted, the default is 0. If M is omitted, the default is
10.
```

```
    * UNSIGNED, if specified, disallows negative values.
    * The UNSIGNED attribute is deprecated for columns of type DECIMAL
    * MySQL stores DECIMAL values in binary format.
```

```
CREATE TABLE emp( sal DECIMAL );
CREATE TABLE emp( sal DECIMAL(5,2) );
```

```
    * Standard SQL requires that DECIMAL(5,2) be able to store any value
with five digits and two decimals, so values that can be stored in the
salary column range from -999.99 to 999.99.
9. FLOAT
    * A small (single-precision) floating-point number.
    * The FLOAT type represent approximate numeric data values.
    * Permissible values are:
        * -3.402823466E+38 to -1.175494351E-38
        * 0
        * 1.175494351E-38 to 3.402823466E+38.
    * These are the theoretical limits, based on the IEEE
      standard. The actual range might be slightly smaller depending on
hardware or operating system.
    * M is the total number of digits and D is the number of digits
following the decimal point.
    * If M and D are omitted, values are stored to the
    limits permitted by the hardware.
    * A single-precision floating-point number is accurate to approximately
7 decimal places.
    * UNSIGNED, if specified, disallows negative values.
    * The UNSIGNED attribute is deprecated for columns of type FLOAT.
    * Size of float is 4 bytes.
10. DOUBLE
    * It is synonym for REAL and DOUBLE PRECISION.
    * The DOUBLE type represent approximate numeric data values.
    * A normal-size (double-precision) floating-point number.
    * Permissible values are:
        * -1.7976931348623157E+308 to -2.2250738585072014E-308
        * 0
        * 2.2250738585072014E-308 to 1.7976931348623157E+308.
    * These are the theoretical limits, based on the IEEE    standard. The
actual range might be slightly smaller depending on hardware or operating
system.
    * A double-precision floating-point number is accurate to approximately
15 decimal places.
    * UNSIGNED, if specified, disallows negative values.
    * The UNSIGNED attribute is deprecated for columns of type DOUBLE
    * Size of double is 8 bytes.
```

Date and time types

```
* The date and time data types for representing temporal values are DATE,
TIME, DATETIME, TIMESTAMP, and YEAR.
* Each temporal type has a range of valid values, as well as a “zero” value
that may be used when you specify an invalid value that MySQL cannot
represent.
1. DATE
    * The DATE type is used for values with a date part but no time part.
    * MySQL retrieves and displays DATE values in 'YYYY-MM-DD' format.
    * *  For example, '2012-12-31', '2012/12/31', '2012^12^31', and
'2012@12@31' are equivalent.
    * The supported range is '1000-01-01' to '9999-12-31'.
    * SIZE of DATE is 3 bytes.
2. TIME
    * MySQL retrieves and displays TIME values in 'hh:mm:ss' format.
    * The hours part may be so large because the TIME type can be used not
only to represent a time of day (which must be less than 24 hours), but
also elapsed time or a time interval between two events (which may be much
greater than 24 hours, or even negative).
    * The range is '-838:59:59.000000' to '838:59:59.000000'.
    * MySQL recognizes TIME values in several formats, some of which can
include a trailing fractional seconds part in up to microseconds (6 digits)
precision.
    * Size of TIME is 3 bytes
3. DATETIME
    * The DATETIME type is used for values that contain both date and time
parts.
    * MySQL retrieves and displays DATETIME values in 'YYYY-MM-DD hh:mm:ss'
format.
    * For example, '2012-12-31 11:30:45', '2012^12^31 11+30+45',
'2012/12/31 11*30*45', and '2012@12@31 11^30^45' are equivalent.
    * The date and time parts can be separated by T rather than a space.
For example, '2012-12-31 11:30:45' '2012-12-31T11:30:45' are equivalent.
    * The supported range is '1000-01-01 00:00:00' to '9999-12-31
23:59:59'.
    * Size of DATETIME is 8 bytes
4. YEAR
    * The YEAR type is a 1-byte type used to represent year values.
    * It can be declared as YEAR with an implicit display width of 4
characters, or equivalently as YEAR(4) with an explicit display width.
    * MySQL displays YEAR values in YYYY format, with a range of 1901 to
2155, and 0000
    * Size of year is 1 byte
5. TIMESTAMP
    * The TIMESTAMP data type is used for values that contain both date and
time parts.
    * TIMESTAMP has a range of '1970-01-01 00:00:01' UTC to '2038-01-19
03:14:07' UTC.
    * TIMESTAMP values are stored as the number of seconds since the epoch
('1970-01-01 00:00:00' UTC).
    * A TIMESTAMP cannot represent the value '1970-01-01 00:00:00' because
that is equivalent to 0 seconds from the epoch.
    * Size of TIMESTAMP is 4 bytes.
```

String (character and byte) types

```
* The string data types are CHAR, VARCHAR, BINARY, VARBINARY, BLOB, TEXT,
ENUM, and SET.
* For definitions of character string columns (CHAR, VARCHAR, and the TEXT
types), MySQL interprets length specifications in character units.
* For definitions of binary string columns (BINARY, VARBINARY, and the BLOB
types), MySQL interprets length specifications in byte units.
* The ASCII attribute is shorthand for CHARACTER SET latin1.
* The UNICODE attribute is shorthand for CHARACTER SET ucs2.
1. CHAR
    * CHAR is shorthand for CHARACTER.
    * The length of a CHAR column is fixed to the length that you declare
when you create the table.
    * The length can be any value from 0 to 255.
    * When CHAR values are stored, they are right-padded with spaces to the
specified length.
    * When CHAR values are retrieved, trailing spaces are removed
2. VARCHAR
    * VARCHAR is shorthand for CHARACTER VARYING.
    * MySQL stores VARCHAR values as a 1-byte or 2-byte length prefix plus
data.
    * The length prefix indicates the number of bytes in the value.
    * A VARCHAR column uses one length byte if values require no more than
255 bytes, two length bytes if values may require more than 255 bytes.
    * Values in VARCHAR columns are variable-length strings. The length can
be specified as a value from 0 to 65,535.
    * VARCHAR values are not padded when they are stored.
3. BINARY
    * The BINARY type is similar to the CHAR type, but stores binary byte
strings rather than nonbinary character strings.
    * The permissible maximum length is the same for BINARY as it is for
CHAR.
    * It should be used to store BAR_CODE, QR_CODES, ICONS, DIGITAL
SIGNATURE, FINGER PRINT etc.
4. VARBINARY
    * The VARBINARY type is similar to the VARCHAR type, but stores binary
byte strings rather than nonbinary character strings.
    * The permissible maximum length is the same for VARBINARY as it is for
VARCHAR.
    * It should be used to store small pictures, company logo, retina scan
etc.
* A BLOB is a binary large object that can hold a variable amount of data.
The four BLOB types are TINYBLOB, BLOB, MEDIUMBLOB, and LONGBLOB.
* The four TEXT types are TINYTEXT, TEXT, MEDIUMTEXT, and LONGTEXT.
5. TINYTEXT
    * A TEXT column with a maximum length of 255 characters.
    * Each TINYTEXT value is stored using a 1-byte length prefix that
indicates the number of bytes in the value.
    * Stored outside table and away from row. Only address is stored inside
table.
```

```
6. TEXT
    * A TEXT column with a maximum length of 65,535 characters.
    *  Each TEXT value is stored using a 2-byte length prefix that
indicates the number of bytes in the value.
    * Stored outside table and away from row. Only address is stored inside
table.
7. MEDIUMTEXT
    * A TEXT column with a maximum length of 16,777,215 characters.
    * Each MEDIUMTEXT value is stored using a 3-byte
    length  prefix that indicates the number of bytes in the value.
    * Stored outside table and away from row. Only address is stored inside
table.
8. LONGTEXT
    * A TEXT column with a maximum length of 4,294,967,295 or 4GB
characters.
    * Each LONGTEXT value is stored using a 4-byte length prefix that
indicates the number of bytes in the value.
    * Stored outside table and away from row. Only address is stored inside
table.
    * Used for only those columns that will be used only for storage and
display purpose.
    * It should be used to store remarks, comments, feedback, logs, blogs
resume, experience details, moview review etc.
9. TINYBLOB
    * A BLOB column with a maximum length of 255 bytes.
    * Each TINYBLOB value is stored using a 1-byte length prefix that
indicates the number of bytes in the value.
    * Stored outside table and away from row. Only address is stored inside
table.
10. BLOB
    * A BLOB column with a maximum length of 65,535 bytes.
    * Each BLOB value is stored using a 2-byte length prefix that indicates
the number of bytes in the value.
    * Stored outside table and away from row. Only address is stored inside
table.
11. MEDIUMBLOB
    * A BLOB column with a maximum length of 16,777,215 bytes.
    * Each MEDIUMBLOB value is stored using a 3-byte length prefix that
indicates the number of bytes in the value.
    * Stored outside table and away from row. Only address is stored inside
table.
12. LONGBLOB
    * A BLOB column with a maximum length of 4,294,967,295 or 4GB bytes.
    * Each LONGBLOB value is stored using a 4-byte length
    prefix that indicates the number of bytes in the value.
    * Stored outside table and away from row. Only address is stored inside
table.
    * It should be used to store photograph, map, graph, chart, sound,
music, video etc.
13. ENUM
    * An enumeration.
    * Syntax:
        ENUM('value1','value2',...)
    * A string object that can have only one value, chosen
    from the list of values 'value1', 'value2', ..., NULL or the special ''
error value.
    * ENUM values are represented internally as integers.
    * An ENUM column can have a maximum of 65,535 distinct elements.
    * The maximum supported length of an individual ENUM element is M <=
255
    * It requires 1 or 2 bytes, depending on the number of enumeration
values.
```

```
CREATE TABLE shirts
(
    name VARCHAR(40),
    size ENUM('x-small', 'small', 'medium', 'large', 'x-large')
);
INSERT INTO shirts
(name, size)
    VALUES
('dress shirt','large'),
('t-shirt','medium'),
('polo shirt','small');
```

```
14. SET
    * A SET is a string object that can have zero or more values, each of
which must be chosen from a list of permitted values specified when the
table is created.
    * A SET column can have a maximum of 64 distinct members.
```

```
CREATE TABLE myset (col SET('a', 'b', 'c', 'd'));
INSERT INTO
myset(col)
VALUES
('a,d'),
('d,a'),
('a,d,a'),
('a,d,d'),
('d,a,d');
```

```
    * For a value containing more than one SET element, it does not matter
what order the elements are listed in when you insert the value. It also
does not matter how many times a given element is listed in the value. When
the value is retrieved later, each element in the value appears once, with
elements listed according to the order in which they were specified at
table creation time.
```

Day 1

```
* Data, processing, Information
* Database
* Need Of Database
* Applications
* DBMS vs RDBMS
* MySql Introduction
```

Day 2

- Databases do not understand C/C++/Java language. It can understand only SQL("Sequel")[Structured Query Language].
- Initial name of SQL was RQBE(Relational Query By Example). It is introduced by IBM in 1975.
- ANSI is responsible for standardzing SQL. It means that SQL is common for all databases.
- In 2005, code of sql was rewritten in Java.
- Sub Division of SQL commands

1. Data Definition Language[DDL]
    - CREATE
    - ALTER
    - RENAME
    - DESCRIBE
    - DROP
    - TRUNCATE
2. Data Manipulation Language[DML]
    - INSERT
    - UPDATE
    - DELETE
3. Data Query Language[DQL]
    - SELECT
4. Transaction Control Language[TCL]
    - COMMIT
    - ROLLBACK
    - SAVEPOINT
5. Data Control Language[DCL]
    - GRANT
    - REVOKE

Naming Conventions for identifier[Database/ Table / Column name]

1. Maximum 30 characters are allowed
2. Name must begin with character

3. It can contain[A-Z,a-z,0-9,$,_]
4. Reservered word can not be used as identifier.
5. Identifiers are case insensitive.

```
mysql -u root -pmanager
```

```
SELECT user();
SELECT user() FROM DUAL;
```

- DUAL is single row and 2 column table.
- It is a dummy table name, we should use it situations where no tables are referenced:

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

- Comments

```
-- SELECT 5*5 FROM DUAL;
# SELECT 5*5 FROM DUAL;
/* SELECT 5*5 FROM DUAL; */
```

- CURRENT_USER() is a function which returns user name and host name combination for the MySQL account that the server used to authenticate the current client.
- Check exsting Users

```
SELECT User from user;
```

- Creating New User

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

- Delete User

```
DROP USER 'dac'@'localhost';
```

```
DELETE
FROM user
WHERE
User='dac';
```

- Check User Permissions

```
SHOW GRANTS FOR 'dac'@'localhost';
```

- Assigning permission's to user

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

```
FLUSH PRIVILEGES;
```

- Removing permission's to user

```
-- REVOKE INSERT ON *.* FROM 'jeffrey'@'localhost';

-- REVOKE ALL PRIVILEGES
REVOKE ALL
ON
*.*
FROM
'dac'@'localhost';
```

```
FLUSH PRIVILEGES;
```

```
SHOW GRANTS FOR 'dac'@'localhost';
```

```
-- mysql -u dac -p
mysql -u dac -pdac
```

- In context of SQL, database is also called as schema.
- List databases:

```sql
SHOW DATABASES;
```

- Create new database

```sql
-- CREATE DATABASE dac_db;  -- or
CREATE SCHEMA dac_db;
```

- Check currenlty selected database

```sql
-- SELECT DATABASE() -- or
SELECT DATABASE() FROM DUAL;
```

```sql
-- SELECT SCHEMA() -- or
SELECT SCHEMA() FROM DUAL;
```

- Working with database

```sql
USE dac_db;
SELECT DATABASE() FROM DUAL;
```

- List tables from database

```sql
SHOW TABLES;
```

- Creating Table
- Syntax: CREATE TABLE tbl_name (col_name column_definition);

```sql
CREATE TABLE books
(
    id INT,
    name VARCHAR(50),
    author VARCHAR(50),
    subject VARCHAR(50),
    price FLOAT
);
```

```
SHOW TABLES;
```

- CHECK TABLE STRUCTURE

```
DESCRIBE books; -- or
DESC books; -- or
EXPLAIN books;  -- or
SHOW COLUMNS FROM books;
```

- Rename Table

```
RENAME TABLE books TO books_tbl;
```

- If we want to modify table structure then we should use ALTER statement.
- How to rename column?
    - Syntax: ALTER TABLE tbl_name RENAME COLUMN old_col_name TO new_col_name;

```
ALTER TABLE books
RENAME COLUMN id TO book_id;

ALTER TABLE books
RENAME COLUMN name TO book_name;

ALTER TABLE books
RENAME COLUMN author TO author_name;

ALTER TABLE books
RENAME COLUMN subject TO subject_name;
```

- How to modify column Definition?
    - Syntax ALTER TABLE tbl_name MODIFY [COLUMN] col_name column_definition.

```
ALTER TABLE books
MODIFY COLUMN book_name VARCHAR(256 );

ALTER TABLE books
MODIFY author_name VARCHAR(256 );

ALTER TABLE books
MODIFY price DOUBLE;

ALTER TABLE books
MODIFY book_id INT(5);
```

- How to add new column in table?
  - Syntax: ALTER TABLE tbl_name ADD [COLUMN] col_name column_definition

```
ALTER TABLE books
ADD COLUMN pub_name VARCHAR(50);
```

- How to drop column from table?
  - Syntax: ALTER TABLE tbl_name DROP [COLUMN] col_name

```
ALTER TABLE books
DROP COLUMN pub_name;
```

- How to insert record into table?
  - Syntax: INSERT INTO tbl_name (col_name1 , col_name2 ...) VALUES (value_list)

```
INSERT INTO books
( book_id, book_name, author_name, subject_name, price )
VALUES
( 1, 'Let Us C', 'Yashwant Kanetkar', 'C', 450 );
```

```
INSERT INTO books
VALUES
( 2, "More Effective C++", "Scott Mayers", "C++", 550 );
```

```
INSERT INTO books
(book_id, subject_name, book_name, author_name, price )
VALUES
( 3, 'Java','Java Certification', 'Khalid Mughal', 650 );
```

```
INSERT INTO books
(book_id, book_name , price )
VALUES
( 4, 'CLR Via C#', 850 );
```

```
INSERT INTO books
VALUES
( 5, 'OS Concepts',null,NULL,500 );
```

```sql
INSERT INTO books()VALUES( );
```

```sql
INSERT INTO books
VALUES
(6,'The C Prog Lang.','Dennis Ritche','C', 450),
(7,'C++ Complete Reference','Herbert Schildt', 'C++', 600),
(8,'Java Head First', 'Kathy Siera','Java', 800);
```

- How to view records/rows?
  - SELECT is used to retrieve rows selected from one or more tables
  - Syntax SELECT FROM table_references;

```sql
SELECT * FROM books;
```

- How to create copy of table?

```sql
CREATE TABLE new_books
AS
SELECT * FROM books;
```

```sql
CREATE TABLE new_books_tbl
AS
SELECT book_id, subject_name, book_name, author_name, price FROM books;
```

- How to copy table structure?

```sql
CREATE TABLE book_table LIKE books;
```

```sql
INSERT INTO book_table
( SELECT * FROM books );
```

- How to import sql file?

```sql
CREATE DATABASE classwork;
USE classwork;
```

```
--SOURCE (Drag and Drop ) .sql file here
SOURCE /Users/sandeepkulange/Desktop/DBT/classwork-db.sql;
```

- Fetch records from all columns

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

```
SELECT
*
FROM
books;
```

- Fetch records from few columns

```
SELECT
name, author, price
FROM
books;
```

```
SELECT
-- name, author, price, price + price * 0.10
-- name, author, price, price + price * 0.10 AS Final_Price
-- name, author, price, price + price * 0.10 AS "Final Price"
-- name, author, price, price + price * 0.10 AS 'Final Price'
-- name, author, price, price + price * 0.10 'Final Price'
name Name, author Author, price Price, price + price * 0.10 'Final Price'
FROM
books;
```

- DISTINCT

```
-- SELECT  author  FROM books;
SELECT DISTINCT author  FROM books;
```

- LIMIT

```
-- SELECT * FROM books LIMIT 4;
SELECT * FROM books LIMIT 4,3;
```

```
SELECT
*
FROM books
-- ORDER BY price ASC;
-- ORDER BY price;
ORDER BY price DESC;
```

```
SELECT
*
FROM books
-- ORDER BY subject, author;
-- ORDER BY subject ASC, author DESC;
ORDER BY subject DESC, author ASC;
```

```
SELECT
*
FROM books
-- ORDER BY subject;
ORDER BY 4;
```

```
SELECT
id, name, author, price, price + price * 0.10 Final
FROM books
-- ORDER BY Final;
-- ORDER BY Final DESC;
ORDER BY 5 DESC;
```

- How to delete table?
    - DROP TABLE removes one or more tables.
    - Syntax: DROP TABLE tbl_name;
    - It is a DDL statement, which removes table data as well as table structure.

```
DROP TABLE new_books_tbl;
DROP TABLE book_table, new_books;
```

- How to truncate records?
    - TRUNCATE TABLE empties a table completely.
    - Logically, TRUNCATE TABLE is similar to a DELETE statement that deletes all rows, or a sequence of DROP TABLE and CREATE TABLE statements.

   - Syntax : TRUNCATE TABLE tbl_name;

```
TRUNCATE TABLE books;
```

Day 3

Where clause

```
mysql -u dac -p
mysql -u dac -pdac
mysql -u dac -pdac classwork
```

- System Information Functions

1. DATABASE()

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

2. USER

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

3. VERSION

```
SELECT VERSION() FROM DUAL;
```

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

- If we want filter rows then we should use where clause.
- Operators
   - Arithmetic Operators : ( ), / ,* ,+, -
   - Relational Operators : <, <=, >, >=, ==, != / <>
   - BETWEEN, IN, LIKE
   - Logical Operators : AND, OR, NOT
- Get record of books from "books" table whose subject is "Java Programming"

```
SELECT * FROM books
WHERE subject='Java Programming';
```

```
SELECT * FROM books
WHERE subject="Java Programming";
```

```
SELECT * FROM books
WHERE subject="java programming";
```

- Find book(s) of 'Dennis Ritchie' from 'books' table;

```
SELECT * FROM books
WHERE author='Dennis Ritchie';
```

```
SELECT * FROM books
WHERE id=3001;
```

- Find book(s) of 'Herbert Schildt' Whose price is greater than 500.

```
SELECT * FROM books;

SELECT * FROM books
WHERE author='Herbert Schildt';

SELECT * FROM books
WHERE author='Herbert Schildt' AND price > 500;
```

- List book(s) of 'Operating Systems' or list book(s) whose author s 'Herbert Schildt' and price is greater than 500.

```
SELECT * from books;

SELECT * from books
WHERE subject='Operating Systems';

SELECT * FROM books
WHERE author='Herbert Schildt' AND price > 500;

SELECT * FROM books
```

```
WHERE subject='Operating Systems' OR author='Herbert Schildt' AND price >
500;

SELECT * FROM books
WHERE subject='Operating Systems' OR ( author='Herbert Schildt' AND price
> 500);

SELECT * FROM books
WHERE NOT subject='Operating Systems' OR author='Herbert Schildt' AND
price > 500;

SELECT * FROM books
WHERE  ( subject='Operating Systems' OR author='Herbert Schildt' )AND
price > 500;

SELECT * FROM books
WHERE  NOT ( subject='Operating Systems' OR author='Herbert Schildt' )AND
price > 500;
```

- Find employee(s) from EMP table whose manager is NULL

```
SELECT * FROM EMP;

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

- Find employee(s) from EMP table whose comm is not NULL

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

- Find employee(s) from EMP table whose sal is in between 2000 and 3000.

```
SELECT * FROM emp
-- WHERE sal >= 2000 AND sal <= 3000;
WHERE sal BETWEEN 2000 AND 3000;
```

- Find employee(s) from EMP table whose sal is in not between 2000 and 3000.

```
SELECT * FROM emp
WHERE sal NOT BETWEEN 2000 AND 3000;
```

```
SELECT * FROM emp
WHERE ename BETWEEN 'D' AND 'M';
```

- We can use BETWEEN operator to compare number, date and string.

```
SELECT * FROM emp;

SELECT * FROM emp
WHERE deptno=10;

SELECT * FROM emp
WHERE deptno=20;

SELECT * FROM emp
WHERE deptno=30;

SELECT * FROM emp
WHERE deptno=40;
```

- List all the employee(s) whose dept no is either 10, 20 or 30

```
SELECT * FROM emp
WHERE deptno=10 OR deptno=20 OR deptno=30;
```

```
SELECT * FROM emp
-- WHERE deptno IN (10, 20, 30 );
WHERE deptno NOT IN (10, 20, 30 );
```

- Find Details of Employee which is having third highest salary.

```
SELECT * FROM emp;

SELECT * FROM emp
ORDER BY sal DESC;

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

- If want to search values in column using specific pattern then we should use LIKE operator.
- ☐)
- Print Details of Java Books whose name starts with 'Java'.

```sql
SELECT * FROM books;

SELECT * FROM books
WHERE name LIKE 'Java%';
```

- Print Details of Java Books whose name ends with 'Language'.

```sql
SELECT * FROM books;

SELECT * FROM books
WHERE name LIKE '%Language';
```

Print Details of Java Books whose name contains 'Programming'.

```sql
SELECT * FROM books;

SELECT * FROM books
WHERE name LIKE '%Programming%';
```

```sql
SELECT * FROM books;

SELECT * FROM books
WHERE name LIKE '%in%';
```

```sql
SELECT * FROM books;

SELECT * FROM books
WHERE name LIKE 'J%e';
```

```sql
SELECT * FROM books
WHERE name LIKE '_N%';
```

```sql
SELECT * FROM books
WHERE name LIKE '%e_';
```

Update Statement

- If we want to update column definition then we should use ALTER statement.

- If we want to update record/row then we should use UPDATE statement.
- Syntax: UPDATE table_reference SET assignment_list;
- Update book name from books table whose book id is 1026.

```
UPDATE books
SET book_name = 'Linux Programming Interface'
WHERE book_id = 1026;
```

- Update subject name and author name from books table whose book id is 1026.

```
UPDATE books
SET subject_name='OS', author_name='Michael Kerrisk'
WHERE book_id=1026;
```

- Update price of C++ books by 5%.

```
UPDATE books
SET price=price - price * 0.05
WHERE subject_name='C++';
```

- Update price of all books by 5%.

```
UPDATE books
SET price = price - price * 0.05;
```

Delete Statement

- DELETE is a DML statement that removes rows from a table.
- Syntax : DELETE FROM tbl_name;
- Delete book from table whose book id is 1016.

```
DELETE FROM books
WHERE book_id = 1016;
```

- Delete all the books from table whose subject name is 'AWP'

```
DELETE FROM books
WHERE subject_name='AWP';
```

- Delete all the books from table

```
DELETE FROM books;
```

Function

- If we want to process data stored in row then we can use function.
- Types:
  1. Single Row Function
  2. Multi Row Function / Group Function / Aggregate Function
- Functions
  - Single Row Functions
    - String Functions (URL: http://dev.mysql.com/doc/refman/8.0/en/string-functions.html) 1 length : LENGTH(str) 2 concat : CONCAT(str1,str2,...) 3 upper : UPPER(str) 4 lower : LOWER(str) 5 left : LEFT(str,len) 6 right : RIGHT(str,len) 7 substr SUBSTRING(str,pos), SUBSTRING(str FROM pos), SUBSTRING(str,pos,len), SUBSTRING(str FROM pos FOR len) 8 lpad : LPAD(str,len,padstr) 9 rpad : RPAD(str,len,padstr) 10 ltrim : LTRIM(str) 11 rtrim : RTRIM(str) 12 trim : TRIM([{BOTH | LEADING | TRAILING} [remstr] FROM] str), TRIM([remstrFROM] str) 13 replace : REPLACE(str, FROM_str, TO_STR) 14 reverse : REVERSE(str) 15 instr : INSTR(str,substr) 16 ascii : ASCII(str) 17 char 18 bin : BIN(N)

    - Numeric Functions(URL: http://dev.mysql.com/doc/refman/8.0/en/mathematical-functions.html)

      1. Round : ROUND(X), ROUND(X,D)
      2. Truncate : TRUNCATE(X,D)
      3. Floor : FLOOR(X)
      4. Ceil : CEILING(X)
      5. Sign : SIGN(X)
      6. Mod : MOD(N,M), N % M, N MOD M
      7. Sqrt : SQRT(X)
      8. Power : POW(X,Y)
      9. Abs : ABS(X)
      10. Log : LOG(X), LOG(B,X)
      11. Sin : SIN(X), cos, tan

    - Date Functions(URL: http://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html)

      1. sysdate : SYSDATE([fsp])

      2. Now : NOW([fsp])

      3. Sleep : SLEEP(duration)

      4. Adddate : ADDDATE(date,INTERVAL expr unit), ADDDATE(expr,days)

      5. Date_add: DATE_ADD(date,INTERVAL expr unit), DATE_SUB(date,INTERVAL expr unit)

6. Datediff : DATEDIFF(expr1,expr2)
7. Lastday
8. Dayname
9. Monthname
10. Day, month, year
11. add-time
12. Date()
13. Time()
14. date_format

- List Functions(URL: http://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html)

  1. Greatest
  2. Least
  3. Isnull
  4. Strcmp

- Control flow functions(https://dev.mysql.com/doc/refman/8.0/en/control-flow-functions.html)

  1. if()
  2. Ifnull
  3. nullif()

- Multi Row Functions
  - Group / Aggregate functions(https://dev.mysql.com/doc/refman/8.0/en/group-by-functions.html) 1.sum 2.Avg 3.min 4.max 5.count

1. Can not use nonaggregated/ regular column with group function.
2. Can not use single row function with group function.
3. Can not use group function in where clause.
4. Can not nest group function.

Day 4

Group By

- If we want to display statistics / reports then we can take help of group by.
- Find out total count of jobs in emp table.

```
SELECT COUNT(job) FROM emp;
```

- Find out jobwise count;

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

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

- If we want to use column with group function then either that should be aggregate column or it must be present in group by.
    - SELECT COUNT(job) FROM emp GROUP BY job; //OK
    - SELECT job, COUNT(job) FROM emp GROUP BY job; //OK
- If we use any column in group by then it is not necessary to present in select query but column present in select must be exist inside group by.
    - SELECT deptno, COUNT(job) FROM emp GROUP BY job; //NOT OK
    - SELECT COUNT(job) FROM emp GROUP BY job, deptno; //OK
    - SELECT deptno, COUNT(job) FROM emp GROUP BY job, deptno; //OK
- Find out total count of department.

```
SELECT COUNT(deptno) FROM emp;
SELECT DISTINCT deptno FROM emp;
```

- Find out count of employees departmentwise.

```
SELECT COUNT(empno) FROM emp;

SELECT COUNT(empno) FROM emp
GROUP BY deptno;

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

- Find out departmentwise, jobwise count.

```
SELECT COUNT(job)
FROM emp;

SELECT COUNT(job)
FROM emp
GROUP BY job;

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

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

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

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

- Find out Departmentwise, job wise count of employees from dept no 20.

```
SELECT deptno, job, COUNT(job)
FROM emp
WHERE deptno=20
GROUP BY deptno, job
ORDER BY deptno;
```

- Find out avg salary of employees from emp

```
SELECT avg(sal) FROM emp;
```

- Find out avg salary of employees departmentwise

```
SELECT avg(sal)
FROM emp
GROUP BY deptno;

SELECT deptno, avg(sal)
```

```
FROM emp
GROUP BY deptno;
```

- Find out avg salary of employees departmentwise whose avg sal > 2000.

```
SELECT deptno, avg(sal)
FROM emp
GROUP BY deptno
HAVING AVG(sal) > 2000;
```

- If we want to use group function to check condition then we should use Having clause.
- Display manager ID and number of employees managed by the manager.

```
SELECT manager_id FROM employees;

SELECT DISTINCT manager_id FROM employees;

SELECT COUNT(  EMPLOYEE_ID ) FROM employees;

SELECT COUNT(  EMPLOYEE_ID )
FROM employees
GROUP BY MANAGER_ID;

SELECT MANAGER_ID, COUNT(  EMPLOYEE_ID ) AS Count
FROM employees
GROUP BY MANAGER_ID;
```

- Display average salary of employees in each department who have commission percentage.

```
SELECT AVG(SALARY)
FROM employees;

SELECT DEPARTMENT_ID, AVG(SALARY)
FROM employees
GROUP BY DEPARTMENT_ID;

SELECT DEPARTMENT_ID, AVG(SALARY)
FROM employees
GROUP BY DEPARTMENT_ID
HAVING COMMISSION_PCT > 0;  -- NOT OK
```

- We can not use regular/non aggregate columns in Having clause.

```
SELECT DEPARTMENT_ID, AVG(SALARY)
FROM employees
```

```
WHERE COMMISSION_PCT > 0
GROUP BY DEPARTMENT_ID;
```

- Display job ID, number of employees, sum of salary, and difference between highest salary and lowest salary of the employees of the job.

```
SELECT SUM(SALARY)
FROM employees;

SELECT
SUM(SALARY) "Total Salary",
MAX( SALARY) "Highest Salary",
MIN(SALARY) "Lowest Salary",
MAX( SALARY) - MIN(SALARY) "Difference"
FROM employees;

SELECT
JOB_ID,
SUM(SALARY) "Total Salary",
MAX( SALARY) "Highest Salary",
MIN(SALARY) "Lowest Salary",
MAX( SALARY) - MIN(SALARY) "Difference"
FROM employees
GROUP BY JOB_ID;
```

SQL Transaction

- If we execute DML statements/queries as a single unit then it is called transaction.
- COMMIT, ROLLBACK AND SAVEPOINT are TCL(Transaction Control Language) commands. These are used for SQL transaction.
- User 1

```
-- Step 1
SELECT * FROM DEPT;
-- Step 2
INSERT INTO DEPT VALUES( 50, 'QA', 'PUNE');
-- Step 3
SELECT * FROM DEPT;
```

- User 2

```
-- Step 2
SELECT * FROM DEPT;
-- Step 4
SELECT * FROM DEPT;
```

* By default, all DML statements execute in auto commit mode. If we want to disable auto commit mode the should use "START TRANSACTION" statement.

```
START TRANSACTION;
-- Execute DML1
-- Execute DML2
-- Execute DML3
COMMIT;
```

```
START TRANSACTION;
-- Execute DML1
-- Execute DML2
-- Execute DML3
ROLLBACK;
```

```
START TRANSACTION;
-- Execute DML1
SAVEPOINT PT1;
-- Execute DML2
SAVEPOINT PT2;
-- Execute DML3
ROLLBACK TO PT2;
COMMIT;
```

* User 1

```
-- Step 1
START TRANSACTION;
-- Step 2
INSERT INTO DEPT VALUES(50, 'QA', 'PUNE');
-- Step 3
SELECT * FROM DEPT;
-- Step 5
COMMIT WORK;
-- Step 6
SELECT * FROM DEPT;
```

* User 2

```
-- Step 4
SELECT * FROM DEPT;
```

```
-- Step 7
SELECT * FROM DEPT;
```

- User 1

```
-- Step 1
START TRANSACTION;
-- Step 2
DELETE FROM DEPT WHERE DEPTNO=50;
-- Step 3
SELECT * FROM DEPT;
-- Step 5
ROLLBACK WORK;
-- Step 6
SELECT * FROM DEPT;
```

- User 2

```
-- Step 4
SELECT * FROM DEPT;
-- Step 7
SELECT * FROM DEPT;
```

- During transaction, if we execute DDL statement then transaction gets commited automatically.
- In Case of run time problems(power cut off, automatic restart) transaction gets rollbacked.

Row locking

- Types:
  1. Optimistic Locking
     - If User2 is trying to modifying record which is User1 is already modified in transaction then row gets clocked for User2. It is called Optimistic locking. When User1 will commit/rollback transaction then row will available for User2.
     - User 1

```
-- STEP 1
START TRANSACTION;
-- STEP 2
UPDATE DEPT SET loc='NEW DELHI'
WHERE deptno=10;
-- STEP 5
COMMIT;
-- STEP 6
SELECT * FROM DEPT;
-- STEP 10
SELECT * FROM DEPT;
```

```
    * User 2
```

```sql
-- STEP 2
START TRANSACTION;
-- STEP 4
UPDATE DEPT SET loc='NAGPUR'
WHERE deptno=10;    -- BLOCK
-- STEP 7
SELECT * FROM DEPT;
-- STEP 8
COMMIT;
-- STEP 9
SELECT * FROM DEPT;
```

```
2. Pesimistic Locking
* In this case using "For UPDATE" we can lock records before performing
DML operations.
* Row will be unlocked after execution of ROLLBACK/COMMIT.
* User1
```

```sql
-- STEP 1
START TRANSACTION;
-- STEP 3
SELECT * FROM DEPT WHERE deptno=10
FOR UPDATE;
-- STEP 5
UPDATE DEPT SET loc='NEW YORK'
WHERE deptno=10;
-- STEP 6
SELECT * FROM DEPT;
-- STEP 7
COMMIT;
-- STEP 11
SELECT * FROM DEPT;
```

```
* User2
```

```sql
-- STEP 2
START TRANSACTION;
```

```
-- STEP 4
UPDATE DEPT SET loc='NEW DELHI'
WHERE deptno=10;    //BLOCKED
-- STEP 8
SELECT * FROM DEPT;
-- STEP 9
COMMIT;
-- STEP 10
SELECT * FROM DEPT;
```

- SQL Transaction support ACID property.

1. Atomic : Operation should be perfored both side of transaction or it should be rollbacked.
2. Consitent : Database should display same data at both side of transaction
3. Isolation : Every transaction should run seperatly.
4. Durable : After completion of transaction all the changes should be reflected in database.

Query Performance

- If we want check query performance then we should use "EXPLAIN FORMAT=JSON".
- query_cost attribute contains value which describe execution time.

```
EXPLAIN FORMAT=JSON select * from emp;
```

```
EXPLAIN FORMAT=JSON SELECT deptno, AVG(sal)
FROM emp
GROUP BY deptno
HAVING DEPTNO=20;
```

```
EXPLAIN FORMAT=JSON SELECT deptno, AVG(sal)
FROM emp
WHERE DEPTNO=20
GROUP BY deptno;
```

Index

- It is a SQL feature which is used for faster searching.
- Syntax: CREATE INDEX index_name ON tbl_name (key_part,...);
- key_part: {col_name [(length)] | (expr)} [ASC | DESC]
- Types of index:
    1. Normal Index
    2. Composite Index
    3. Unique Index
- Normal Index

```
CREATE INDEX idx_dept_deptno
ON
dept( deptno);  -- ASC : by default
-- dept( deptno ASC)
-- dept( deptno DESC )
```

```
CREATE INDEX idx_dept_dname
ON
dept( dname);
```

```
CREATE INDEX idx_dept_loc
ON
dept( loc);
```

```
SHOW INDEXES FROM dept;
```

- Syntax to drop index
    - DROP INDEX index_name ON tbl_name

```
DROP INDEX idx_dept_loc ON dept;
DROP INDEX idx_dept_dname ON dept;
DROP INDEX idx_dept_deptno ON dept;
```

- Composite Index
    - We can create index on multiple columns. It is caled composite index.

```
CREATE INDEX idx_dept_dname_loc
ON
dept(dname, loc);
```

- Unique Index
    - Column in index contains only unique value.

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

```
CREATE UNIQUE INDEX idx_dept_dname_loc
ON
dept(dname, loc);
```

- Limitations
    - If we perform DML operations on table then DB engine need to update index. Hence it degrades performance.
    - Index creation is a slower process. If we create index then internally complex data structure is used (BTree,HashTable) which takes time.
    - It takes space on HDD.

Day 5

Constraint

- If we want store accurate data inside table then we should use constraints
- To store accurate data, we need to put/apply some restrictions on column. It is called constraint.
- We can apply constraints during table creation or after table creatiion using ALTER statement.
- SQL supports following constraints
    1. DEFAULT
    2. NOT NULL
    3. UNIQUE
    4. CHECK
    5. PRIMARY KEY
    6. FOREIGN KEY
- We can apply these constraints at two levels:
    1. Column Level
    2. Table Level

Default constraint

```
CREATE TABLE address
(
    city VARCHAR(50),
    state VARCHAR(50),
    pincode CHAR(6),
    country VARCHAR( 50 ) DEFAULT "India"
);
```

```
INSERT INTO address
VALUES
('Pune','MH','411057','India');

SELECT * FROM address
```

```
INSERT INTO address
(city, state,pincode)
VALUES
('Mumbai','MH','411001');
SELECT * FROM address;
```

```
INSERT INTO address
VALUES
('Mumbai','MH','411001',NULL);
SELECT * FROM address;
```

- Remove Default constraint

```
ALTER TABLE address
ALTER country DROP DEFAULT;
```

- SET Default constraint

```
ALTER TABLE address
ALTER country SET DEFAULT "India";
```

```
INSERT INTO address
(city, state,pincode)
VALUES
('Nasik','MH','415001');
SELECT * FROM address;
```

NOT NULL Constraint

- If we dont want to store NULL value inside record/row then we should use NOT NULL constraint on column.

```
CREATE TABLE employees
(
    id INT NOT NULL,
    name VARCHAR(50),
    salary FLOAT
);
```

```
INSERT INTO
employees
VALUES( 1, 'ABC', 10000);
```

```
INSERT INTO
employees
```

```
VALUES( NULL, 'PQR', 20000); -- Error
```

```
INSERT INTO
employees
(name, salary)
VALUES
( 'PQR', 20000); -- Error
```

```
UPDATE employees
SET id = NULL   -- Error
WHERE name='ABC';
```

- How to remove NOT NULL contraint

```
ALTER TABLE employees
MODIFY COLUMN id INT;

DESC employees;

INSERT INTO
employees
(name, salary)
VALUES
( 'PQR', 20000); -- OK
```

- Set NOT NULL constraint on column

```
ALTER TABLE employees
MODIFY COLUMN id INT NOT NULL;
```

UNIQUE Constraint

- If we want store unique value inside record/row then we should use UNIQUE constraint.

```
CREATE TABLE employees
(
    id INT UNIQUE,
    name VARCHAR(50),
    salary FLOAT,
    email VARCHAR(50),
    UNIQUE(email)
);
DESC employees;
```

```
INSERT INTO employees
VALUES
( 1, 'Sandeep', 45000.50, 'sandeep@gmail.com');
```

```
INSERT INTO employees
VALUES
(2, 'Soham', 65000.50, 'sandeep@gmail.com');
-- NOT OK
```

```
INSERT INTO employees
VALUES
(1, 'Soham', 65000.50, 'soham@gmail.com');
-- NOT OK
```

```
INSERT INTO employees
VALUES
(NULL, 'Soham', 65000.50, 'soham@gmail.com'); -- OK
```

- Table can contain multiple UNIQUE keys.
- Unique key can be NULL.

```
CREATE TABLE student
(
    roll INT,
    `div` CHAR(10),
    name VARCHAR(50),
    marks FLOAT,
    CONSTRAINT UC_STUDENT_ROLL_DIV UNIQUE(roll, `div`)
);
```

```
INSERT INTO
student
VALUES
( 1, 'A', 'Sandeep', 54.0 ),
( 1, 'B', 'Rahul', 80.0 ),
( 2, 'A', 'Ketan', 75.0 ),
( 2, 'B', 'Akash', 60.0 );
```

```
INSERT INTO
student
VALUES
( 1, 'A', 'Nilesh', 65.0 );
```

- Remove Unique Key Constraint

```
ALTER TABLE student
DROP INDEX UC_STUDENT_ROLL_DIV;
```

- Set Unique Key Constraint

```
ALTER TABLE student
ADD CONSTRAINT UC_STUDENT_ROLL_DIV UNIQUE(roll, `div`);
```

```
CREATE TABLE employee
(
    id INT,
    name VARCHAR(50),
    salary FLOAT,
    CONSTRAINT UC_EMPLOYEE_ID UNIQUE( ID )
);
```

CHECK Constraint

- It is introduced in MySQL 8.0.16.
- It is used to check range of values.

```
CREATE TABLE employees
(
    id INT NOT NULL UNIQUE,
    name VARCHAR(50) UNIQUE,
    salary FLOAT CHECK( salary > 0 ),
);
```

```
CREATE TABLE employees
(
    id INT NOT NULL UNIQUE,
    name VARCHAR(50) UNIQUE,
    salary FLOAT,
    CHECK( salary > 0 )
);
```

```
CREATE TABLE employees
(
    id INT NOT NULL UNIQUE,
    name VARCHAR(50) UNIQUE,
    salary FLOAT,
    CONSTRAINT CHK_EMPLOYEES_SALARY CHECK( salary > 0 )
);
```

```
INSERT INTO employees
VALUES( 1, 'ABC', 5000 );

INSERT INTO employees
VALUES( 2, 'PQR', -5000 ); --ERROR

INSERT INTO employees
VALUES( 2, 'PQR', NULL );
```

- Remove Check Constraint

```
ALTER TABLE employees
DROP CHECK CHK_EMPLOYEES_SALARY;

INSERT INTO employees
VALUES( 3, 'XYZ', -5000 );
```

- Set Check Constraint

```
ALTER TABLE employees
ADD CONSTRAINT CHK_EMPLOYEES_SALARY
CHECK( salary BETWEEN 8000 AND 20000 );
```

```
INSERT INTO employees
VALUES
(1, 'Sandeep', 15000 );
-- (1, 'Sandeep', 25000 ); -- NOT OK
-- (1, 'Sandeep', 5000 ); --NOt OK
```

PRIMARY KEY CONSTRAINT

- PRIMARY KEY = UNIQUE + NOT NULL
- Per table we can declare only one primary key.
- It can be single value or composite value.
- Primary key do not contain NULL value.
- If we want to idenitfy record in a table uniquely then we should delare primary key.
- Types of primary Key
    1. Natural Primary key
    2. Composite Primary Key
    3. Surrogate Primary Key
- If single value is sufficient to identify record in a table uniquely then it is called Natural primary key.
- Example
    - Customers
        - name
        - address
        - contact_number
        - email[PK]
    - Employee
        - name
        - empid[PK]
        - salary
        - dept
        - designation
- When single value is insufficient to identify record in a table uniquely then we can use combination of values as a primary key. It is called composite primary key.
- Example
    - Student
        - name
        - roll
        - div
        - marks
        - primary key[roll + div]
- If we use AUTO_GENERATED value from column as a primary key then it is called Surrogate primary key
- Example
    - Student
        - regid [AUTO_INCREMENT] [PK]
        - name
        - email
- Primary Key declaration at column level

```
CREATE TABLE employees
(
    empid INT PRIMARY KEY,
    name VARCHAR(50),
    salary FLOAT
);
```

- Primary Key declaration at table level

```
CREATE TABLE employees
(
    empid INT,
    name VARCHAR(50),
    salary FLOAT,
    -- PRIMARY KEY( empid ) -- OK
    CONSTRAINT PK_EMPLOYEES_EMPID PRIMARY KEY( empid )
);
```

```
INSERT INTO employees
VALUES( 1, 'ABC', 10000); -- OK

INSERT INTO employees
VALUES( 1, 'PQR', 20000); -- NOT OK

INSERT INTO employees
VALUES( NULL, 'PQR', 20000); -- NOT NULL
```

- Composite Primary Key declaration at table level

```
CREATE TABLE student
(
    roll INT,
    division VARCHAR(50),
    name VARCHAR(50),
    marks FLOAT,
    CONSTRAINT PK_STUDENT_ROLL_DIVISION PRIMARY KEY( roll, division )
);
```

```
INSERT INTO student
VALUES
( 1, 'A', 'AAA', 50 ),
( 2, 'A', 'BBB', 50 ),
( 3, 'B', 'CCC', 50 ),
( 4, 'B', 'DD', 50 );
```

```
INSERT INTO student
VALUES
( 1, 'A', 'EEE', 60 ); --Not OK
```

```
INSERT INTO student
VALUES
( 1, NULL, 'EEE', 60 ); --Not Ok
```

```
INSERT INTO student
VALUES
( NULL, 'A', 'EEE', 60 ); --Not Ok
```

Surrogate Primary Key declaration at table level

```
CREATE TABLE orders
(
    id INT AUTO_INCREMENT,
    status VARCHAR( 50 ),
    CONSTRAINT PK_ORDERS_ID PRIMARY KEY( id)
);
```

```
INSERT INTO orders
(status)
VALUES('Pending');
```

```
INSERT INTO orders
(status)
VALUES('Delivered');
```

```
ALTER TABLE orders AUTO_INCREMENT = 100;
INSERT INTO orders
(status)
VALUES('Pending');
```

- Drop primary key

```
ALTER TABLE employees
DROP PRIMARY KEY;
```

- SET primary key

```
ALTER TABLE employees
ADD CONSTRAINT PK_EMPLOYEES_EMPID PRIMARY KEY( empid );
```

FOREIGN KEY Constraint

- If parent-child relationship exist between the table then we need to use foreign key.
- column in child table which referes to column in parent table is called foreign key.
- A table, which contains, foreign key, is called child table. A table which contains candidate key is called parent table.

```
CREATE TABLE dept
(
    dno INT,
    dname VARCHAR(50),
    loc VARCHAR( 50 ),
    CONSTRAINT PK_DEPT_DNO PRIMARY KEY(dno)
);

INSERT INTO dept
VALUES
( 10, 'DEV', 'PUNE'),
( 20, 'QA', 'MUMBAI');
```

```
CREATE TABLE emp
(
    eno INT,
    ename VARCHAR( 50 ),
    sal FLOAT,
    dno INT,
    -- FOREIGN KEY dno REFERENCES dept(dno)
    CONSTRAINT FK_EMP_DEPT_DNO FOREIGN KEY ( dno)  REFERENCES dept( dno )
);

INSERT INTO emp
VALUES
( 1, 'Rahul', 10000, 10),
( 2, 'Amit', 15000, 10),
( 3, 'Sandeep', 5000, 20);
```

```
INSERT INTO emp
VALUES
(4,'Ketan',7000,30);
```

```
INSERT INTO emp
VALUES
(4,'Ketan',7000,NULL);
```

- Remove foreign key

```
ALTER TABLE emp
DROP FOREIGN KEY FK_EMP_DEPT_DNO;

DESC emp;

INSERT INTO emp
VALUES
(4,'Ketan',7000,30);
```

- Set foreign key

```
ALTER TABLE emp
ADD CONSTRAINT FK_EMP_DEPT_DNO FOREIGN KEY ( dno)  REFERENCES dept( dno );
```

```
DELETE FROM dept
WHERE dno=20; -- NOT OK
```

```
DELETE FROM emp
WHERE eno=3; --OK
DELETE FROM dept
WHERE dno=20; -- NOT OK
```

- In case of foreign key, if we want to delete record from parent table then it is necessary to delete all records associated with primary key from child table.

```
ALTER TABLE emp
DROP FOREIGN KEY FK_EMP_DEPT_DNO;

ALTER TABLE emp
ADD CONSTRAINT FK_EMP_DEPT_DNO FOREIGN KEY ( dno)  REFERENCES dept( dno )
ON UPDATE CASCADE ;
```

```
UPDATE dept
SET dno=1
WHERE dname='DEV';

SELECT * FROM DEPT;
SELECT * FROM EMP;
```

```
ALTER TABLE emp
DROP FOREIGN KEY FK_EMP_DEPT_DNO;

ALTER TABLE emp
ADD CONSTRAINT FK_EMP_DEPT_DNO FOREIGN KEY ( dno)  REFERENCES dept( dno )
ON DELETE CASCADE ;
```

```
DELETE FROM dept
WHERE dno=1;
SELECT * FROM DEPT;
SELECT * FROM EMP;
```

```
ALTER TABLE emp
DROP FOREIGN KEY FK_EMP_DEPT_DNO;

ALTER TABLE emp
ADD CONSTRAINT FK_EMP_DEPT_DNO FOREIGN KEY ( dno)  REFERENCES dept( dno )
ON DELETE CASCADE ON UPDATE CASCADE ;
```

- foreign_key_checks is system variable that we can use to enable or disable foreign key.

```
SHOW VARIABLES;

SHOW VARIABLES LIKE 'f%';

SELECT @@foreign_key_checks from dual;

INSERT INTO emp
VALUES
( 1, 'Rahul', 10000, 10),
( 2, 'Amit', 15000, 10),
( 3, 'Sandeep', 5000, 20); -- NOt OK
```

```
SET @@foreign_key_checks = 0;
SELECT @@foreign_key_checks from dual;
```

```
INSERT INTO emp
VALUES
( 1, 'Rahul', 10000, 10),
( 2, 'Amit', 15000, 10),
( 3, 'Sandeep', 5000, 20);

SET @@foreign_key_checks = 1;
SELECT @@foreign_key_checks from dual;

INSERT INTO dept
VALUES
( 10, 'DEV', 'PUNE'),
( 20, 'QA', 'MUMBAI');
```

Joins

- If we insert duplicate records inside table then it takes more space on HDD. To minimize this wastage we should use normalization.
- In case of normalization, we can create multiple tables and we can maintain relationship between them using constraints(PK , FK).
- If we want to get data from multiple tables then we should join.
- Types of joins
    1. Cross join
    2. Inner Join
    3. Outer Join
        1. Left Outer Join
        2. Right Outer Join
        3. Full Join
    4. Self Join

Cross Join

- In case cross join, a table which is having maximum records is called "driving table" and another table is called "driven" table.
- If we want to combine/join each record of driving table with all the records of driven table then we should use cross join.
- It is also called as cartesion join.

```
emplist : collection of emp objects
deptlist : collection of dept objects
for( emp : emplist )
{
    for( dept : deptlist )
    {
        print( emp & dept );
    }
}
```

```
SELECT * FROM
EMP e CROSS JOIN DEPT d;
```

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

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

```
SELECT e.ename, d.dname FROM
EMP e, DEPT d;
```

- It is a join without condition, hence it faster than all other joins.

Inner Join

- If we want fetch matching records from both tables then we should use inner join

```
for( e : emplist )
{
    for( dept d : deptlist )
    {
        if( e.deptno == d.deptno )
            print( e & d );
    }
}
```

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

```
SELECT e.ename, d.dname
FROM emp e , dept d
WHERE e.deptno=d.deptno;
```

LEFT OUTER JOIN

- If we want to get all the records (matching & non matching) from left table and matching records from right table then we should use left outer join.

```
for( e : emplist )
{
    flag = 0;
    for( d : deptlist )
    {
        if( e.dno == d.dno )
        {
            flag = 1;
            print( e & d )
        }
    }
    if( flag == 0 )
    print( e & NULL );
}
```

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

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

RIGHT OUTER JOIN

- If we want to get all the records (matching & non matching) from right table and matching records from left table then we should use right outer join.

```
for(d  : deptlist )
{
    flag = 0;
    for( e : emplist )
    {
        if( d.dno == e.dno )
        {
            flag = 1;
            print( e, d );
        }
    }
    if( flag == 0 )
     print( NULL, d );
}
```

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

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

View

Temporary Table

Day 6

Cross Join

```sql
SELECT e.ename, d.dname
FROM emp e CROSS JOIN dept d;
```

```sql
SELECT e.ename, d.dname
FROM emp e, dept d;
```

Inner Join

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

```sql
SELECT e.ename, d.dname
FROM emp e , dept d
WHERE e.deptno = d.deptno;
```

LEFT OUTER JOIN

```sql
SELECT e.ename, d.dname
FROM emp e LEFT OUTER JOIN dept d
ON e.deptno = d.deptno;
```

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

RIGHT OUTER JOIN

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

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

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

Full Join

- MySQL do not support full join but using set operators we can implement it.
- Set Operators
    1. UNION
        - It supress duplicate element
    2. UNION ALL
        - It doesn't supress duplicate element
- Full join is union of left outer join and right outer join

```
SELECT e.ename, d.dname
FROM emp e LEFT JOIN dept d
ON e.deptno = d.deptno
UNION
SELECT e.ename, d.dname
FROM emp e RIGHT JOIN dept d
ON e.deptno = d.deptno;
```

```
SELECT e.ename, d.dname
FROM emp e LEFT JOIN dept d
ON e.deptno = d.deptno
UNION ALL
SELECT e.ename, d.dname
FROM emp e RIGHT JOIN dept d
ON e.deptno = d.deptno;
```

SELF JOIN

- If we use join on same table themn it is called self join.
- Display name of employee and its manager

```
for( e : emp )
{
    for( m : emp )
    {
        if( e.mgr == m.empno )
            print( e.ename & m.ename )
    }
}
```

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

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

```
SELECT e.ename, m.ename
FROM emp m , emp e
WHERE e.mgr = m.empno;
```

Table Relations

1. One To One
    - emp(1) <--->(1)address
2. One To Many
    - dept(1) <--->(*)emp
3. Many To one
    - emp(*) <---> (1)dept
4. Many To Many
    - emp() <---->() meeting

Tables

1. dept : deptno, dname

```
CREATE TABLE dept
(
    deptno INT,
    dname VARCHAR(50),
    CONSTRAINT PK_DEPT_DEPTNO
    PRIMARY KEY(deptno)
);
INSERT INTO dept
VALUES
( 10,'DEV'),
( 20,'QA'),
( 30,'ACCOUNT'),
( 40,'OPERATIONS');
```

2. emp : empno, ename, deptno

```
CREATE TABLE emp
(
    empno INT,
    ename VARCHAR(50),
    deptno INT,
    CONSTRAINT PK_EMP_EMPNO PRIMARY KEY(empno),
    CONSTRAINT FK_EMP_DEPT FOREIGN KEY ( deptno ) REFERENCES dept( deptno
)
);

INSERT INTO emp
VALUES
(1, 'Rahul', 10),
(2, 'Amit', 10),
(3, 'Sandeep', 20),
(4, 'Nilesh', 50),
(5, 'Nitin', 50);
```

3. address : loc, pincode, empno

```
CREATE TABLE address
(
    id INT AUTO_INCREMENT,
    loc VARCHAR(50),
    pincode CHAR(6),
    empno INT,
    CONSTRAINT PK_ADDRESS_ID PRIMARY KEY(id), CONSTRAINT FK_ADDRESS_EMP
FOREIGN KEY ( empno ) REFERENCES emp( empno )
);
INSERT INTO address
( loc, pincode, empno )
VALUES
('Karad','11111', 1),
('Kolhapur','22222', 2),
('A.Nagar','33333', 3),
('Junnar','44444', 4),
('Panchgani','55555', 5);
```

4. meeting : meetno, topic, venue

```
CREATE TABLE meeting
(
    meetno INT,
    topic VARCHAR(50),
    venue VARCHAR(50),
    CONSTRAINT PK_MEETING_MEETNO PRIMARY KEY ( meetno )
);
INSERT INTO meeting
VALUES
( 100, 'Scheduling', 'Directors Cabin'),
( 200, 'Annual Meet', 'Board Room'),
( 300,'App Desgin','Co-Directors Cabin');
```

5. emp_meeting : meetno, empno

```
CREATE TABLE emp_meeting
(
    id INT AUTO_INCREMENT,
    meetno INT,
    empno INT,
    CONSTRAINT PK_EMP_MEETING_ID PRIMARY KEY ( id ),
    CONSTRAINT FK_EM_MEETING FOREIGN KEY ( meetno ) REFERENCES meeting(
meetno ),
    CONSTRAINT FK_EM_EMP FOREIGN KEY ( empno ) REFERENCES emp( empno )
);
INSERT INTO emp_meeting
( meetno, empno )
VALUES
( 100, 3),
( 100, 4),
( 100, 5),
( 200, 1),
( 200, 2),
( 200, 3),
( 200, 4),
( 200, 5),
( 300, 1 ),
( 300, 2 ),
( 300, 5 );
```

- Print meeting topic and emp names attending that meeting

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

SELECT em.meetno, em.empno, e.ename
FROM emp_meeting em INNER JOIn emp e
ON em.empno = e.empno;
```

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

- Print meeting topic and employee names and their addresses

```


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

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

SELECT m.meetno, m.topic, em.empno, e.ename,a.loc
FROM meeting m INNER JOIN emp_meeting em
ON m.meetno = em.meetno
INNER JOIN emp e
ON em.empno = e.empno
INNER JOIN address a
ON e.empno = a.empno;
```

- Print meeting topic and enames and their depts

```
SELECT m.meetno, m.topic, em.empno, e.ename, d.dname
FROM meeting m INNER JOIN emp_meeting em
ON m.meetno = em.meetno
INNER JOIN emp e
ON em.empno = e.empno
INNER JOIN dept d
ON e.deptno = d.deptno;
```

- Print meeting topic and employee names and their depts as well as addresses

```
SELECT m.topic, e.ename, a.loc, d.dname
FROM meeting m INNER JOIN emp_meeting em
ON m.meetno = em.meetno
INNER JOIN emp e
ON em.empno = e.empno
INNER JOIN address a
ON e.empno = a.empno
```

```
INNER JOIN dept d
ON e.deptno = d.deptno;
```

- Write a query that lists each order number followed by the name of the customer who made the order.

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

- Write a query that gives the names of both the salesperson and the customer for each order along with the order number.

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

- Write a query that produces all customers serviced by salespeople with a commission above 12%. Output the customer's name, the salesperson's name, and the salesperson's rate of commission.

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

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

```
SELECT c.cname, c.rating, o.onum, s.sname, o.amt * s.comm AS Commision
FROM CUSTOMERS c INNER JOIN ORDERS o
ON c.cnum = o.cnum
INNER JOIN SALESPEOPLE s
ON c.snum = s.snum
WHERE c.rating > 100;
```

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

```
SELECT s1.sname, s1.city, s2.sname, s2.city
FROM SALESPEOPLE s1 INNER JOIN SALESPEOPLE s2
ON s1.city = s2.city
```

```
 WHERE s1.snum > s2.snum;
--WHERE s1.sname < s2.sname;
```

- Write a query that produces the names and cities of all customers with the same rating as Hoffman.

```
SELECT c1.cname, c1.city, c1.rating
FROM CUSTOMERS c1 INNER JOIN CUSTOMERS c2
ON c1.rating = c2.rati  ng
WHERE c2.cname = 'Hoffman' AND c1.cname != 'Hoffman';
```

Sub Query

- If we write select query inside select query then it is called sub query.
- If we sub query returns single row then it is called single row sub query.
- Find all emps whose sal is more than avg sal of emps.

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

-- 2073.214286
SELECT * FROM emp WHERE sal > 2073.214286;

SET @AVG_SAL = 2073.214286;
SELECT * FROM emp WHERE sal > @AVG_SAL;

SET @AVG_SAL = ( SELECT AVG(sal) FROM emp );
SELECT * FROM emp WHERE sal > @AVG_SAL;

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

- Find emp with 3rd highest sal

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

SET @SAL = 2975.00;
SELECT * FROM emp WHERE sal = @SAL;

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

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

- Display emps whose sal is more than sal of all salesman

```
SELECT * FROM emp;
SELECT sal FROM emp WHERE job='salesman';
SELECT MAX(sal) FROM emp WHERE job='salesman';

SET @SAL = 1600.00;
SELECT * FROM emp WHERE sal > @SAL;

SET @SAL = ( SELECT MAX(sal) FROM emp WHERE job='salesman' );
SELECT * FROM emp WHERE sal > @SAL;

SELECT * FROM emp WHERE sal > ( SELECT MAX(sal) FROM emp WHERE
job='salesman' );
```

- If sub query returns multipe rows then such query is called multi row sub query.

```
SELECT * FROM emp WHERE sal > ALL ( SELECT sal FROM emp WHERE
job='salesman' );
```

- IF we want to process multi row sub query then we should use ALL, ANY and IN operator.
- ALL : equivalent to logical AND
- ANY : equivalent to logocal OR
- IN : To check range of values
- Display emps whose sal is more than sal of any salesman;

```
SELECT * FROM emp;
SELECT sal FROM emp WHERE job ='salesman';
SELECT MIN(sal) FROM emp WHERE job ='salesman';

SET @SAL = ( SELECT MIN(sal) FROM emp WHERE job ='salesman' );
SELECT * FROM emp WHERE sal > @SAL;


SELECT * FROM emp WHERE job != 'salesman' AND sal > ( SELECT MIN(sal) FROM
emp WHERE job ='salesman' ) ;

SELECT * FROM emp WHERE job != 'salesman' AND sal >  ANY( SELECT sal FROM
emp WHERE job ='salesman' ) ;
```

- Display depts in which emps are there

```
SELECT * FROM dept;
SELECT deptno from emp;
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 );
```

Co-Related Sub Query

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


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

- If sub query depends of outer query then it is called co related sub query.
- Display depts in which emps exists

```
SELECT * FROM dept d
WHERE exists ( SELECT deptno from emp e WHERE e.deptno = d.deptno );
```

- Display depts in which emps doesn't exists

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

- Delete emps from emp table whose sal is less than avg(sal) of emp.

```
SELECT AVG(sal) from emp;

DELETE FROM emp
WHERE sal < ( SELECT AVG(sal) from emp );
-- Not OK
```

- We can not perform DML operations on table which is used in sub query.

```
DELETE FROM dept d
WHERE NOT EXISTS ( SELECT deptno from emp e WHERE e.deptno = d.deptno );
-- OK
```

Day 7

VIEW

- DCL Commands are GRANT and REVOKE.

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

GRANT ALL ON *.* TO 'dmc'@'localhost' WITH GRANT OPTION;
```

```
REVOKE ALL ON *.* FROM 'dmc'@'localhost';
```

- View is SQL feature which is used to give controlled access to the columns for user.
- View is based on select query.
- Syntax:

```
CREATE VIEW view_name [(column_list)]
AS select_statement;
```

- Create copy of emp table;

```
CREATE TABLE employees
AS ( SELECT * FROM emp );

SELECT * FROM employees;
```

- Create view to get all the records from employees table.

```
CREATE VIEW view1
AS ( SELECT * FROM employees );

SHOW TABLES;
SHOW FULL TABLES;
DESC employees;
DESC view1;
```

- Insert record using view1

```
INSERT INTO view1
( empno, ename, sal, deptno)
VALUES
(  7935, 'Sandeep', 8000, 30 ); -- OK
```

- Fetch records using view1

```
SELECT * FROM view1;
SELECT * FROM employees;
```

- View do not store data/record rather it stores pointer of table which is used to store data.
- View contains only select queries.
- Create view using specific columns

```
CREATE VIEW view2
AS ( SELECT empno, ename, sal, deptno FROM employees );

SHOW FULL TABLES;
DESC view2;
SELECT * FROM view2;
```

- Insert record into view2

```
INSERT INTO view2
VALUES( 7936, 'Rahul', 5500, 20 );

SELECT * FROM view2;
SELECT * FROM employees;
```

- Create view with computed column

```
CREATE VIEW view3
AS
( SELECT empno, ename, deptno, sal, sal * 12 "Annual Salary" FROM employees
);

SHOW FULL TABLES;
DESC view3;
SELECT * FROM view3;
```

- Insert record into view3

```
INSERT INTO view3
VALUES( 7937, 'Ketan', 5000, 10, 60000 );
-- Column 'Annual Salary' is not updatable
```

- Create view using where clause

```
CREATE VIEW view4
AS
( SELECT empno, ename, deptno, sal FROM employees WHERE sal > 2000 );

SHOW FULL TABLES;
DESC view4;
SELECT * FROM view4;
```

- Insert record into view4

```
INSERT INTO view4
VALUES( 7937, 'Ketan', 1500, 10 );

SELECT * FROM view4;
SELECT * FROM employees;
```

- Create View using "WITH CHECK OPTION"

```
CREATE VIEW view5
AS
( SELECT empno, ename, deptno, sal FROM employees WHERE sal > 2000 )WITH
CHECK OPTION;

SHOW FULL TABLES;
DESC view5;
SELECT * FROM view5;
```

- Insert record into view5

```
INSERT INTO view5
VALUES( 7937, 'Ketan', 1500, 10 );
```

- Create view using group by

```
CREATE VIEW view6
AS
```

```
(SELECT subject, SUM(price) "Total price" FROM books GROUP BY subject );

SHOW FULL TABLES;
DESC view6;
SELECT * FROM view6;
```

- Insert record into view6

```
INSERT INTO view6
VALUES('C Sharp', 2000);
```

- Create view using joins

```
CREATE VIEW view7
AS
( SELECT e.ename, d.dname
  FROM emp e INNER JOIN dept d
  ON e.deptno = d.deptno
);

SHOW FULL TABLES;
DESC view7;
SELECT * FROM view7;
```

- Insert into view7

```
INSERT INTO view7
( ename, dname )
VALUES( 'ABC', 'QA'); -- Not OK
```

- Create joins using sub query

```
CREATE VIEW view8
AS
(
    SELECT * FROM emp
    WHERE sal > ALL( SELECT sal FROM emp WHERE job='SALESMAN')
);

SHOW FULL TABLES;
DESC view8;
SELECT * FROM view8;
```

- Insert record into view8;

```
INSERT INTO view8
( empno, ename, sal, deptno )
VALUES( 7935,'Sandeep',5000,30); --Not OK
```

- Drop view

```
DROP VIEW view1;
DROP VIEW view2;
DROP VIEW view3;
DROP VIEW view4;
DROP VIEW view5;
DROP VIEW view6;
DROP VIEW view7;
DROP VIEW view8;

-- DROP VIEW view1, view2, view3, view4, view5, view6, view7, view8;
```

Types of view

- There are 2 types of view:
    1. Simple view
        - It allows us to perform DML (INSERT,UPDATE,DELETE)operations
    2. Complex view
        - A view which contains computed column, group by, joins and sub queries in select query then it is called complex view.
    3. Complex view do not allows us to perform DML operations.

Advantages

```
1. If we write view on table then no need write SQL statement multiple
times.
In other words we can achive reusability.
2. View reduces complexity hence it is simple to use.
3. It allows to give access to limited columns as well as to perform
limited operations. In other words, we can achive security.
```

VIEW Security

```
CREATE VIEW view1
AS ( SELECT empno, ename, sal, deptno FROm employees);

SELECT user from user; -- Error
SELECT user from mysql.user; -- OK
```

```
use mysql;
SELECT DATABASE() FROM DUAL;
SELECT user from user;
```

- Create dmc user

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

- Assign permission

```
GRANT SELECT, INSERT, UPDATE, DELETE
ON classwork.* TO 'divesd'@'localhost' WITH GRANT OPTION;
```

- Create dmc user

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

- Use 'divesd'@'localhost' user

```
mysql -u divesd -pdivesd;
SELECT USER() FROM DUAL;
SHOW DATABASES;
USE classwork;
SELECT DATABASE() FROM DUAL;
SHOW TABLES;
```

- Assign permissions to precat.

```
mysql -u divesd -pdivesd;
GRANT SELECT ON classwork.view1 TO 'precat'@'localhost';

SHOW DATABASES;
USE classwork;
SHOW FULL TABLES;
DESC view1;
SELECT * FROM view1;

INSERT INTO view1
VALUES(7937,'Ketan',5000,20);
```

Temporary Table

- If we want to store result returned by select query temporarily then we should create Temporary table.
- It gets space on HDD.
- We can use Temporary table for current session only.
- If we logout or if connection fail then Temporary gets detroy.
- We can give same name to the base table and Temporary table. In this is preference will be given to Temporary table. But it is not recommended.
- Different can create Temporary table with same name but it is not accessible to each other.
- Create Temporary table

```
CREATE TEMPORARY TABLE tbl_name( );
```

```
mysql -u dac -pdac classwork;
CREATE TEMPORARY TABLE new_employees
LIKE  employees;

DESC new_employees;
SHOW TABLES;
INSERT INTO new_employees ( SELECT * FROM employees );
SELECT * FROM new_employees;
EXIT;
```

```
CREATE TEMPORARY TABLE new_employees
AS ( SELECT empno, ename, sal, sal * 12 Total FROM employees );

DESC new_employees;

SELECT SUM( Total ) FROM new_employees;
EXIT;
```

```
CREATE TEMPORARY TABLE new_employees
(
    id INT PRIMARY KEY,
    name VARCHAR( 50 )
);
INSERT INTO new_employees VALUES(1,'DAC');
INSERT INTO new_employees VALUES(1,'DMC');
```

```
CREATE TEMPORARY TABLE new_emp
AS SELECT * FROM emp;

CREATE TEMPORARY TABLE new_dept
```

```sql
AS SELECT * FROM dept;

SELECT e.ename, d.dname
FROM new_emp e INNER JOIN new_dept d
ON e.deptno = d.deptno;

CREATE VIEW view1
AS ( SELECT e.ename, d.dname
FROM new_emp e INNER JOIN new_dept d
ON e.deptno = d.deptno );
```

Day 8

MySQL Programming

- SQL Statements:
  - DDL : CREATE, ALTER, RENAME, DESCRIBE, DROP, TRUNCATE
  - DML : INSERT, UPDATE, DELETE
  - DQL : SELECT
  - TCL : COMMIT, ROLLBACK, SAVEPOINT
  - DCL : GRANT, REVOKE
- Programming Constructs
  - if
  - if-else
  - else if
  - switch case
  - loop(do-while, while, for)
- MySQL Programming = SQL Statements + Programming constructs
- PL/SQL is concept of ORACLE which is used for database programming.
- MySQL Programming is also called as Persistent Stored Module(PSM).
- In 1992, ANSI accepted PSM.
- PSM consist of stored procedure, function, trigger,cursor etc
- PSM contains programming constructs :
  - IF, IF-ELSE, ELSEIF, CASE, WHILE, REPEAT-UNTIL, LABLED LOOP etc.
- If we want to improve query performance then we can use PSM.

Stored Procedure

- In SQL/ MySQL, procedure is a function which do not return any value.
- Stored procedure get space on server HDD. Server save it into compiled form.
- If we want create stored procedure then we should use follwing syntax:

```
DELIMITER $$
CREATE PROCEDURE sp_name (params type)
BEGIN
    //SQL Statement1;
    //SQL Statement2;
END $$
DELIMITER ;
```

```
DELIMITER $$
CREATE PROCEDURE sp_insert_emp( )
BEGIN
    INSERT INTO emp (empno, ename, sal, deptno )
    VALUES( 7935,'ABC',5000, 30);
```

```
END $$
DELIMITER ;
```

- List procedure from database.

```
SHOW PROCEDURE STATUS;

pager less -SFX;
SHOW PROCEDURE STATUS WHERE db='classwork';

SHOW PROCEDURE STATUS LIKE 'sp%';
```

- Drop stored procedure

```
DROP PROCEDURE sp_proc1;
DROP PROCEDURE dac_db.sp_test;
```

- Call stored procedure from terminal /command line.

```
CALL sp_insert_emp( );
SELECT * FROM emp;
```

- First Create .sql File and type following program inside it.

```
DELIMITER $$
CREATE PROCEDURE sp_delete_emp( )
BEGIN
    DELETE FROM emp WHERE empno=7935;
END $$
DELIMITER ;
```

- Include and compile .sql file from terminal

```
-- SOURCE pathname.sql
SOURCE ../proc_1.sql
```

- Call Stored Procedure from terminal

```
CALL sp_delete_emp( );
```

PSM User Defined Variables

- Declare number variable in C.

```
int main( void )
{
    int number;
    return 0;
}
```

- Declare number variable in MySQL.

```
DELIMITER $$
CREATE PROCEDURE sp_proc1( )
BEGIN
    DECLARE number INT;
    DECLARE value INTEGER;
END $$
DELIMITER ;
```

- Initialize number in C

```
int main( void )
{
    int number = 10;
    return 0;
}
```

- Initialize number in MySQL

```
DELIMITER $$
CREATE PROCEDURE sp_proc1( )
BEGIN
    DECLARE number INTEGER DEFAULT 10;
END $$
DELIMITER ;
```

- Assign value 20 to the variable number in C

```
int main( void )
{
    int number;
    number = 20;
    return 0;
}
```

- Assign value 20 to the variable number in MySQL

```
DELIMITER $$
CREATE PROCEDURE sp_proc1( )
BEGIN
    DECLARE number INTEGER;
    SET number = 20;
    -- SELECT 20 INTO number;
END $$
DELIMITER ;
```

```
CREATE TABLE result
(
    value INT
);
```

```
DELIMITER //
CREATE PROCEDURE sp_proc2( )
BEGIN
    DECLARE num1 INTEGER DEFAULT 10;
    DECLARE num2 INTEGER;
    DECLARE num3 INTEGER;
    SET num2 = 20;
    -- SET num3 = 30;
    SELECT 30 INTO num3;

    INSERT INTO result VALUES( num1 );
    INSERT INTO result VALUES( num2 );
    INSERT INTO result VALUES( num3 );
END //
DELIMITER ;
```

- Check whether year is leap year or not in C

```
int main( void )
{
    int year = 2020;
    int days = 28;
    if( year % 4 == 0 )
    {
        days = 29;
    }
    return 0;
}
```

- IF : The IF statement for stored programs implements a basic conditional construct.
- Syntax

```
    IF search_condition THEN
        statement_list
    END IF
```

- Check whether year is leap year or not in MySQL

```
DELIMITER $$
CREATE PROCEDURE sp_proc3( )
BEGIN
    DECLARE year INTEGER DEFAULT 2020;
    DECLARE days INTEGER DEFAULT 28;
    IF year mod 4 = 0 THEN
        SET days = 29;
    END IF;
    INSERT INTO result VALUES ( days );
END $$
DELIMITER ;
```

```
CALL sp_proc3( );
SELECT * from result;
```

- The IF statement can have THEN, ELSE, and ELSEIF clauses, and it is terminated with END IF.

```
IF search_condition THEN
    statement_list
ELSE
    statement_list
END IF
```

```
int main( void )
{
    int year = 2020;
    int days = 0;
    if( year % 4 == 0)
        days = 29;
    else
        days = 28;
    return 0;
}
```

```
DELIMITER $$
DROP PROCEDURE IF EXISTS sp_proc4;
CREATE PROCEDURE sp_proc4( )
BEGIN
    DECLARE year INTEGER DEFAULT 2021;
    DECLARE days INTEGER DEFAULT 0;
    IF year mod 4 = 0 THEN
        SET days = 29;
    ELSE
        SET days = 28;
    END IF;
    INSERT INTO result VALUES( days );
END $$
DELIMITER ;
```

- Else if construct

```
int main( void )
{
    int month = 0;
    int days = 0;
    if( month == 1 )
        days = 31;
    else if( month == 2 )
        days = 28;
    else if( month == 3 )
        days = 31;
    else if( month == 4 )
        days = 30;
    else
        days = 0;
    return 0;
}
```

```
DELIMITER $$
DROP PROCEDURE IF EXISTS sp_proc4;
CREATE PROCEDURE sp_proc4( )
BEGIN
    DECLARE month INTEGER DEFAULT 4;
    DECLARE days INTEGER DEFAULT 0;
    IF month = 1 THEN
        SET days = 31;
    ELSEIF month = 2 THEN
        SET days = 28;
    ELSEIF month = 3 THEN
        SET days = 31;
    ELSEIF month = 4 THEN
        SET days = 30;
    ELSEIF month = 12 THEN
        SET days = 31;
    ELSE
        SET days = 0;
    END IF;
    INSERT INTO result VALUES( days );
END $$
DELIMITER ;
```

```
DELIMITER $$
DROP PROCEDURE IF EXISTS sp_proc4;
CREATE PROCEDURE sp_proc4( )
BEGIN
    DECLARE month INTEGER DEFAULT 3;
    DECLARE days INTEGER DEFAULT 0;
    IF month = 1 OR month = 3 OR month = 5 THEN
        SET days = 31;
    ELSEIF month = 2 THEN
        SET days = 28;
    ELSE
        SET days = 30;
    END IF;
    INSERT INTO result VALUES( days );
END $$
DELIMITER ;
```

- Switch Case

```
int main( void )
{
    int month = 1;
    int days = 0;
    switch( month )
    {
    case 1:
        days = 31;
        break;
    case 2:
        days = 28;
        break;
    case 3:
        days = 31;
        break;
    case 4:
        days = 30;
        break;
    default:
        days = 0;
        break;
    }
    return 0;
}
```

- Case statement in MySQL
- Syntax

```
CASE case_value
    WHEN when_value THEN statement_list
    [WHEN when_value THEN statement_list]
    ...
    [ELSE statement_list]
END CASE
```

```
DELIMITER $$
DROP PROCEDURE IF EXISTS sp_proc5;
CREATE PROCEDURE sp_proc5( )
BEGIN
    DECLARE month INTEGER DEFAULT 3;
    DECLARE days INTEGER DEFAULT 0;
    CASE month
    WHEN 1 THEN SET days = 31;
    WHEN 2 THEN SET days = 28;
    WHEN 3 THEN SET days = 31;
    WHEN 4 THEN SET days = 30;
    ELSE SET days = 0;
    END CASE;
    INSERT INTO result VALUES( days );
END $$
DELIMITER ;
```

- while loop in C

```
int main( void )
{
    int count = 1;
    int res = 0;
    while( count <= 10 )
    {
        res = res + count;
        count = count + 1;
    }
    //print res;
    return 0;
}
```

- While loop in MySQL
- Syntax

```
WHILE search_condition DO
    statement_list
END WHILE;
```

```
DELIMITER $$
DROP PROCEDURE IF EXISTS sp_proc6;
CREATE PROCEDURE sp_proc6( )
BEGIN
    DECLARE count INTEGER DEFAULT 1;
    DECLARE res INTEGER DEFAULT 0;
    WHILE count <= 10 DO
        SET res = res + count;
        SET count = count + 1;
    END WHILE;
    INSERT INTO result VALUES( res );
END $$
DELIMITER ;
```

- do-while construct in C

```
int main( void )
{
    int count = 1;
    int res = 0;
    do
    {
        res = res + count;
        count = count + 1;
    }while( count <= 10 );
    //print res;
    return 0;
}
```

- REPEAT-UNTIL construct in MySQL

```
REPEAT
    statement_list
UNTIL search_condition
END REPEAT
```

```
DELIMITER $$
DROP PROCEDURE IF EXISTS sp_proc6;
CREATE PROCEDURE sp_proc6( )
BEGIN
    DECLARE count INTEGER DEFAULT 1;
    DECLARE res INTEGER DEFAULT 0;
    REPEAT
        SET res = res + count;
        SET count = count + 1;
        UNTIL count > 10
    END REPEAT;
    INSERT INTO result VALUES( res );
END $$
DELIMITER ;
```

- Infinite loop in C

```
int main( void )
{
    int count = 0;
    while( 1 )
    {
        count = count + 1;
        if( count == 10 )
        {
            //print count;
            break;
        }
    }
    return 0;
}
```

- Labeled loop in MySQL
- Syntax:

```
label: LOOP
    statement_list
END LOOP label;
```

```
DELIMITER $$
DROP PROCEDURE IF EXISTS sp_proc7;
CREATE PROCEDURE sp_proc7( )
BEGIN
    DECLARE count INTEGER DEFAULT 0;
    DECLARE res INTEGER DEFAULT 0;
    label:LOOP
        SET res = res + count;
        SET count = count + 1;
        IF count = 10 THEN
            LEAVE label;
        END IF;
    END LOOP label;
    INSERT INTO result VALUES( res );
END $$
DELIMITER ;
```

- We can pass argument to the stored procedure. To catch it must specify parameters to stored procedure.
- Parameter can be:
    1. IN parameter
    2. OUT parameter
    3. INOUT parameter
- Each parameter is an IN parameter by default.

```
DELIMITER $$
DROP PROCEDURE IF EXISTS sp_insert_emp;
CREATE PROCEDURE sp_insert_emp( IN empno INTEGER, IN ename VARCHAR(50), IN
sal FLOAT, IN deptno INTEGER)
BEGIN
    INSERT INTO emp
    (empno, ename, sal, deptno)
    VALUES(empno, ename, sal, deptno);
END $$
DELIMITER ;
```

```
 SET @empno=7951;
 SET @ename='Sandeep';
 SET @sal=7500.50;
 SET @deptno=10;

 SELECT @empno, @ename, @sal, @deptno FROM DUAL;

 CALL sp_insert_emp( @empno, @ename, @sal, @deptno );
```

- OUT Parameter

```
CREATE TABLE accounts
(
    number INTEGER,
    name VARCHAR(50),
    balance FLOAT,
    type VARCHAR(50)
);
INSERT INTO accounts
VALUES( 101, 'Ketan', 75000, 'Saving');

INSERT INTO accounts
VALUES( 101, 'Akash', 30000, 'Current');
```

- Transfer fund

```
DELIMITER $$
CREATE PROCEDURE sp_transfer_fund
( IN srcNumber INTEGER,
IN destNumber INTEGER,
IN amount FLOAT,
OUT srcBalance FLOAT,
OUT destBalance FLOAT )
BEGIN
    UPDATE accounts SET balance = balance - amount WHERE number=srcNumber;

    UPDATE accounts SET balance = balance + amount WHERE
number=destNumber;

    SELECT balance INTO srcBalance FROM accounts WHERE number = srcNumber;

    SELECT balance INTO destBalance FROM accounts WHERE number =
destNumber;
END $$
DELIMITER ;
```

```
SET @srcAccNumber = 101;
SET @destAccNumber = 102;
SET @amount = 15000;

CALL sp_transfer_fund( @srcAccNumber, @destAccNumber, @amount,
@srcBalance, @destBalance );

SELECT @srcBalance "Source Bal." FROM DUAL;
SELECT @destBalance "Dest Bal." FROM DUAL;
```

- INOUT Parameter

```
DELIMITER $$
CREATE PROCEDURE sp_proc( INOUT str VARCHAR(50) )
BEGIN
   SET str=CONCAT('Hello',' ', UPPER(str));
```

```
END $$
DELIMITER ;
```

```
SET @name = 'dac';
CALL sp_proc( @name );
SELECT @name FROM DUAL;
```

- We can call stored procedure from another stored procedure.

```
DELIMITER $$
CREATE PROCEDURE sp_test( OUT str VARCHAR(50) )
BEGIN
    DECLARE name VARCHAR( 50 );
    SET name = 'sunbeam';
    CALL sp_proc( name );
    SET str = name;
END $$
DELIMITER ;
```

- Runtime error is also called as exception.
- During execution of stored procedure and function we may get runtime error/exeception. To handle it we should define error handler in PSM.
- If we want to handle error/exception in PSM then we should DECLARE HANDLER.
- Syntax:

```
DECLARE handler_action HANDLER
FOR condition_value statement;
```

- handler_action:
    - CONTINUE: Execution of the current program continues.
    - EXIT: Execution terminates.
    - UNDO: Not supported.
- Condition_value for DECLARE ... HANDLER indicates the specific condition:
    1. mysql_error_code

```
DECLARE CONTINUE HANDLER FOR 1051
BEGIN
-- body of handler
END;
```

```
2. SQLSTATE
```

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02'
BEGIN
-- body of handler
END;
```

```
3. condition_name
4. SQLWARNING
5. NOT FOUND
```

```
DECLARE CONTINUE HANDLER FOR NOT FOUND
BEGIN
-- body of handler
END;
```

```
6. SQLEXCEPTION
```

```
INSERT into accounts VALUES( 102,'Rahul',50000,'Loan');
```

```
DELIMITER $$
CREATE PROCEDURE sp_insert_account(number INT, name VARCHAR(50), balance
FLOAT, type VARCHAR(50) )
BEGIN
    DECLARE EXIT HANDLER FOR 1062
    BEGIN
        SELECT 'Duplicate Account';
    END;
    INSERT INTO accounts VALUES( number, name, balance, type );
    -- stm1
    -- stm2
    -- stm3

END $$
DELIMITER ;
```

```
DELIMITER $$
CREATE PROCEDURE sp_insert_account(number INT, name VARCHAR(50), balance
FLOAT, type VARCHAR(50) )
```

```
BEGIN
    DECLARE EXIT HANDLER FOR 1062 SELECT 'Duplicate Account';
    INSERT INTO accounts VALUES( number, name, balance, type );
    -- stm1
    -- stm2
    -- stm3

END $$
DELIMITER ;
```

- The DECLARE ... CONDITION statement declares a named error condition, associating a name with a condition that needs specific handling.
- Syntax:

```
DECLARE condition_name CONDITION FOR condition_value
```

```
DECLARE duplicate_entry CONDITION FOR 1062;
```

```
DELIMITER $$
CREATE PROCEDURE sp_insert_account(number INT, name VARCHAR(50), balance
FLOAT, type VARCHAR(50) )
BEGIN
    DECLARE duplicate_entry CONDITION FOR 1062;
    DECLARE CONTINUE HANDLER FOR duplicate_entry SELECT 'Duplicate
Account';
    INSERT INTO accounts VALUES( number, name, balance, type );
END $$
DELIMITER ;
```

```
DELIMITER $$
CREATE PROCEDURE sp_insert_account(number INT, name VARCHAR(50), balance
FLOAT, type VARCHAR(50) )
BEGIN
    DECLARE duplicate_entry CONDITION FOR SQLSTATE '23000';
    DECLARE CONTINUE HANDLER FOR duplicate_entry
    BEGIN
        SELECT 'Duplicate Account';
    END;

    INSERT INTO accounts VALUES( number, name, balance, type );
END $$
DELIMITER ;
```

Day 9

Stored Procedure

```
CREATE TABLE students
(
    roll_number INTEGER,
    name VARCHAR( 50 ),
    marks FLOAT,
    CONSTRAINT PK_STUDENTS PRIMARY KEY(roll_number)
);
```

```
INSERT INTO students
VALUES
(1,'Yogesh',85),
(2,'Rajiv',56),
(3,'Ketan',78),
(4,'Akash',62);
```

- Define stored procedure to insert record into student table

```
DELIMITER $$
CREATE PROCEDURE sp_insert_student(
IN roll_number INTEGER,
IN name VARCHAR(50),
IN marks FLOAT)
BEGIN
    INSERT INTO students
    VALUES
    (roll_number, name, marks);
END $$
DELIMITER ;
```

```
SHOW PROCEDURE STATUS;
SHOW PROCEDURE STATUS WHERE DB='dac_db';
SHOW PROCEDURE STATUS LIKE 'sp%';
```

```
CALL sp_insert_student(5,'Amit',74);
SELECT * from students;

SET @roll_number = 6;
SET @name = 'sandeep';
SET @marks = 45;
CALL sp_insert_student(@roll_number,@name,@marks);
SELECT * from students;
```

- Define stored procedure to update record into student table

```
DELIMITER $$
DROP PROCEDURE IF EXISTS sp_update_student;
CREATE PROCEDURE sp_update_student(
IN pRoll_number INTEGER,
IN pMarks FLOAT)
BEGIN
    UPDATE students
    SET marks = pMarks
    WHERE roll_number = pRoll_number;
END $$
DELIMITER ;
```

```
CALL sp_update_student( 4, 68 );
SELECT * FROM students;
```

- Define stored procedure to DELETE record from student table

```
DELIMITER $$
DROP PROCEDURE IF EXISTS sp_delete_student;
CREATE PROCEDURE sp_delete_student(
IN pRoll_number INTEGER )
BEGIN
    DELETE FROM students
    WHERE roll_number = pRoll_number;
END $$
DELIMITER ;
```

```
CALL sp_delete_student( 3 );
SELECT * FROM students;
```

Handler For Error Code

```
DELIMITER $$
CREATE PROCEDURE sp_insert_student(
IN roll_number INTEGER,
IN name VARCHAR(50),
IN marks FLOAT)
BEGIN
    -- DECLARE CONTINUE HANDLER FOR 1062 SELECT 'Duplicate entry';

    /* DECLARE CONTINUE HANDLER FOR 1062
    BEGIN
        SELECT 'Duplicate entry';
    END; */

    /* DECLARE DUPLICATE_ENTRY CONDITION FOR 1062;
    DECLARE CONTINUE HANDLER FOR DUPLICATE_ENTRY
    BEGIN
        SELECT 'Duplicate entry';
    END; */

    DECLARE status INTEGER DEFAULT 1;
    -- DECLARE CONTINUE HANDLER FOR 1062 SET status = 0;
    DECLARE CONTINUE HANDLER FOR 1062
    BEGIN
        SET status = 0;
    END;
    INSERT INTO students
    VALUES
    (roll_number, name, marks);
END $$
DELIMITER ;
```

Cursor

- Iteration / Traversing is process of visiting element in collection.
- Records retrurned by select query is called result set.
- If we want to visit rows of result set one by one then we should use cursor.
- Cursor is based of SELECT statement.
- SELECT STATEMENT can contain ORDER BY, LIMIT, GROUP BY, JOINS and SUB QUERIES.
- Cursor is a variable which is used to visit each row one by one.
- Syntax:

```
DECLARE cursor_name CURSOR FOR select_statement;
```

- Cursor declaration to access name of each row.

```
DECLARE cur CURSOR FOR SELECT name FROM students;
```

- Cursor declaration to access name and marks of each row.

```
DECLARE cur CURSOR FOR SELECT name, marks FROM students;
```

- Cursor declaration to access all values from each row.

```
DECLARE cur CURSOR FOR SELECT * FROM students;
```

- Syntax to fetch row

```
FETCH cursor_name INTO var_name ...;
```

Steps to use Cursor

1. Declare variables as per requirement.
2. Declare Cursor
3. Declare NOT FOUND handler
4. Open cursor
5. FETCH record into variable
6. Close Cursor

- Declare cursor to get name of all the students from student table.

```
DELIMITER $$
CREATE PROCEDURE sp_fetch_name( INOUT name_list VARCHAR(500) )
BEGIN
    DECLARE finished INTEGER DEFAULT 0;
    DECLARE sname VARCHAR(50) DEFAULT '';
    -- Declare cursor
    DECLARE cur CURSOR FOR SELECT name FROM students;
    -- Declare NOT FOUND Handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET finished = 1;
    -- Open Cursor
    OPEN cur;
    -- define labled loop
    label:LOOP
        FETCH cur INTO sname;
        IF finished = 1 THEN
            LEAVE label;
        END IF;
        SET name_list = CONCAT( sname,',',name_list);
    END LOOP label;
    -- Close Cursor
    CLOSE cur;
END $$
DELIMITER ;
```

```
SET @list = '';
CALL sp_fetch_name( @list );
SELECT @list FROM DUAL;
```

- "OPEN cursor_name" statement execute SELECT statement and it attaches cursor to it(Result Set).
- "FETCH" statement fetches the next row for the SELECT statement associated with the specified cursor and advances the cursor pointer.
- If a row exists, the fetched columns are stored in the named variables.
- The number of columns retrieved by the SELECT statement must match the number of output variables specified in the FETCH statement.
- If no more rows are available, a No Data condition occurs with SQLSTATE value '02000'. To detect this condition, you can set up a handler for it (or for a NOT FOUND condition)
- "CLOSE cursor_name" statement closes a previously opened cursor.
- If not closed explicitly, a cursor is closed at the end of the BEGIN... END block in which it was declared.
- We can define cursor inside PL BLOCK(BEGIN...END).

```
DELIMITER $$
DROP PROCEDURE IF EXISTS sp_fetch_name;
CREATE PROCEDURE sp_fetch_name( INOUT name_list VARCHAR(500) )
BEGIN
    DECLARE finished INTEGER DEFAULT 0;
    DECLARE sname VARCHAR(50) DEFAULT '';
    -- Declare cursor
    DECLARE cur CURSOR FOR SELECT name FROM students;
    -- Declare NOT FOUND Handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET finished = 1;
    -- Open Cursor
    OPEN cur;
    -- define labled loop
    label:LOOP
        FETCH cur INTO sname;
        IF finished = 1 THEN
            LEAVE label;
        END IF;
        SET name_list = CONCAT( sname,',',name_list);
    END LOOP label;
    -- Close Cursor
    CLOSE cur;
    -- define another cursor

    SET finished = 0;
    -- Open Cursor
    OPEN cur;
    -- define labled loop
    label:LOOP
        FETCH cur INTO sname;
        IF finished = 1 THEN
            LEAVE label;
        END IF;
        SET name_list = CONCAT( sname,',',name_list);
    END LOOP label;
    -- Close Cursor
    CLOSE cur;

END $$
DELIMITER ;
```

```
SET @list = '';
CALL sp_fetch_name( @list );
SELECT @list FROM DUAL;
```

Characteristics of cursor

1. It is forward only. we can skip row or we can not move back.
2. It is read only.
3. MySQL cursors are asensitive cursors. i.e during traversing, if another user make chanages into the table then cursor returns updated data.

- During traversing if we dont want to update the record then we shoud use FOR UPDATE in SELECT query.

Stored Function

- It is server side program which is used to achive reusability.
- If predefined/library defined functions are insufficient to execute business logic then we should define function.
- Syntax:

```
CREATE FUNCTION sp_name (params,... )
RETURNS type [NOT] DETERMINISTIC
BEGIN
    routine_body
END
```

- We can not call stored function explicitly using CALL statement. To invoke a stored function, refer to it in an expression.
- Specifying a parameter as IN, OUT, or INOUT is valid only for a PROCEDURE. For a FUNCTION, parameters are always regarded as IN parameters.

```
DELIMITER $$
CREATE FUNCTION TOUPPERCASE( str VARCHAR(50)) RETURNS VARCHAR( 50 )
DETERMINISTIC
```

```
BEGIN
    DECLARE temp VARCHAR(50) DEFAULT '';
    SET temp = CONCAT('Hello,',' ', UPPER(str));
    RETURN temp;
END $$
DELIMITER ;
```

```
-- CALL TOUPPERCASE('dac'); -- Error
SELECT TOUPPERCASE('dac') "Name" FROM DUAL;
```

- Function can not return multiple values.
- Types of Function:

1. DETERMINISTIC Function:
    - For same input parameters, if function generates same result then it is called DETERMINISTIC function.
    - BIN(10);
    - 90% functions are DETERMINISTIC
2. NON DETERMINISTIC Function:
    - For same input parameters, if function generates different result then it is called non DETERMINISTIC function.
    - e g: count_exp(joinDate);

Trigger

- A trigger is a named database object that is associated with a table, and that activates when a particular event occurs for the table.
- Syntax:

```
CREATE TRIGGER trigger_name
trigger_time trigger_event
ON tbl_name FOR EACH ROW [trigger_order]
BEGIN
    trigger_body
END
```

- trigger_time:{ BEFORE | AFTER }
- trigger_event:{ INSERT | UPDATE | DELETE }
- trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
- The trigger becomes associated with the table named tbl_name, which must refer to a permanent table. You cannot associate a trigger with a TEMPORARY table or a view.
- Following trigger_event values are permitted:

1. INSERT: The trigger activates whenever a new row is inserted into the table.

2. UPDATE: The trigger activates whenever a row is modified.

3. DELETE: The trigger activates whenever a row is deleted from the table.

- TCL operations are not permitted in trigger.
- It doesnt take any parameter or return any value. NEW and OLD are implicit variables available for trigger.
- If we want to maintain log of DML operations then we should use trigger.

```
CREATE TABLE accounts
(
    number INTEGER,
    name VARCHAR(50),
    balance FLOAT,
    type VARCHAR(50)
);
CREATE TABLE transactions
(
    number INTEGER,
    balance FLOAT,
    type VARCHAR( 50 )
);
INSERT INTO accounts
VALUES
(101,'Amol',85000,'Saving'),
(102,'Rupesh',30000,'Current'),
(103,'Sandeep',185000,'Loan');
```

```
DELIMITER $$
CREATE TRIGGER tgr_insert AFTER INSERT ON accounts FOR EACH ROW
BEGIN
    INSERT INTO transactions
    VALUES( NEW.number, NEW.balance, NEW.type );
END $$
DELIMITER ;
```

```
--SHOW TRIGGERS [FROM db_name] [like_or_where;
SHOW TRIGGERS FROM dac_db;
```

```
DELIMITER $$
CREATE TRIGGER tgr_update AFTER UPDATE ON accounts FOR EACH ROW
BEGIN
    INSERT INTO transactions
    VALUES( OLD.number, OLD.balance, OLD.type );

    INSERT INTO transactions
    VALUES( NEW.number, NEW.balance, NEW.type );
END $$
DELIMITER ;
```

SUNBEAM

Sunbeam Infotech

Exploring New Ideas, Reaching New Heights!!!

Agenda

- Normalization
- Getting ready for Normalization
- UNF
- 1-NF
- 2-NF
- 3-NF
- 4-NF / BCNF
- De-normalization

- Normalization
- DAC Feb 2020 @ Sunbeam Infotech

Normalization

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

Getting ready for Normalization

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

Normalization steps

- Normalization is done for each transaction (one by one).
- Normalization is done stepwise.
  - 1-NF (3 steps)
  - 2-NF (3 steps)
  - 3-NF (3 steps)
  - 4-NF (1 step)
- Not all steps needed for each transaction.

1-NF

- 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.

2-NF

- 4. Only table with composite PK to be examined.
- 5. Those cols 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.

3-NF

- 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 element will be PK of new table.
- 9-b. The PK of new table is to be retained in original table for relationship purposes.

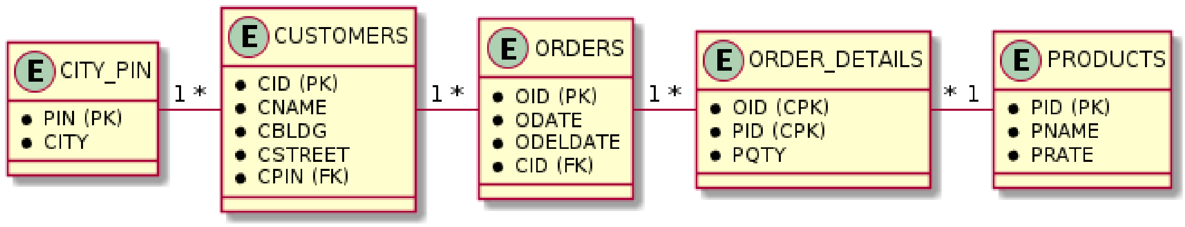
- Repeat steps 7-9 infinitely to examine all non-key elements from all tables and separate them into new table if not dependent on PK.
- This is 3-NF.

4-NF or BCNF

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

ER Diagram

- Entity relationship diagram shows tables and their relations.
- The relations can be
  - One to one
  - One to many
  - Many to one
  - Many to many
- ER diagram also include primary and foreign keys.



De-normalization

- Highly normalized database can cause poor performance for some queries.
- It also leads queries to be very complicated.
- Example:
  - Find total business in last month for each customer.
  - Find city-wise total business of last year.
- To handle these issues some of the rules of Normalization can be compromised.
- The process is known as De-normalization.

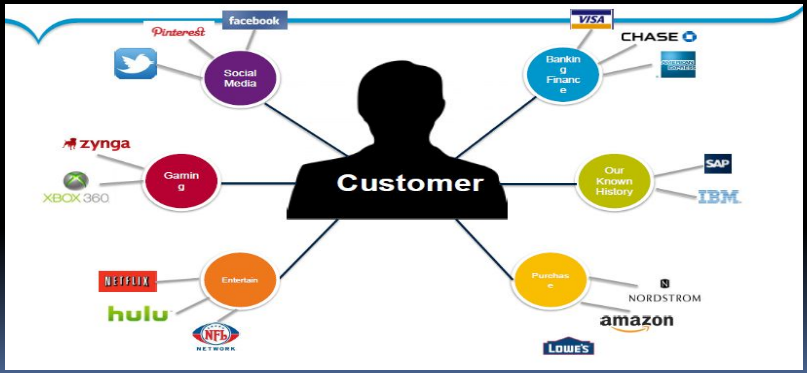
Thank you!

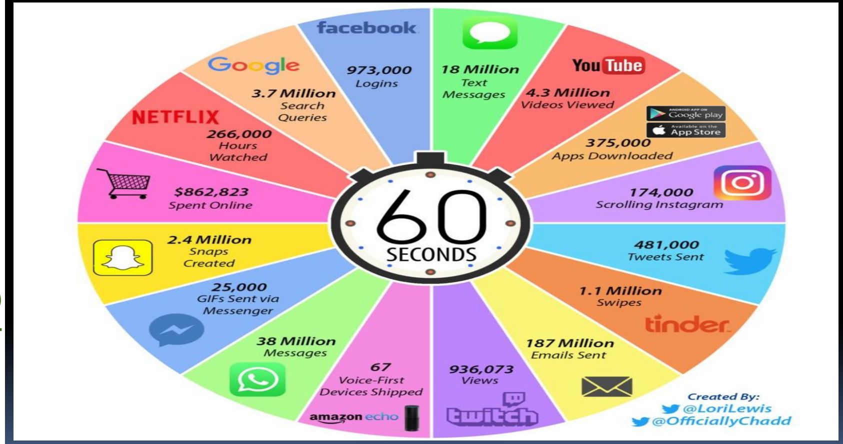
Nilesh Ghule <nilesh@sunbeaminfo.com>

Sunbeam Infotech

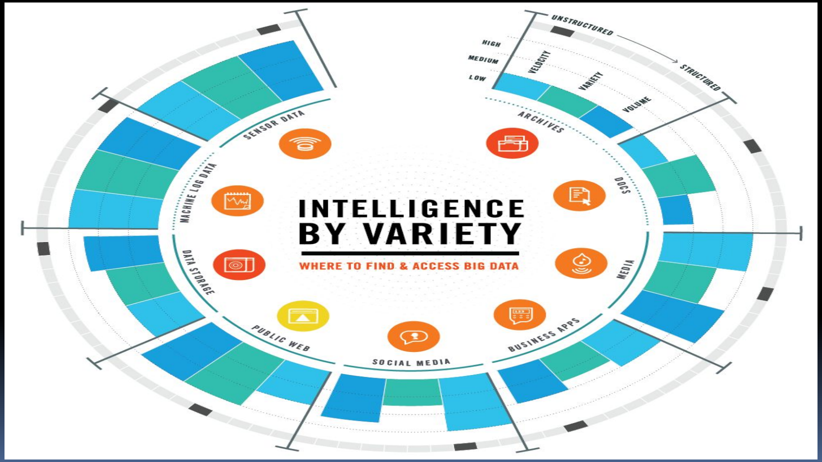
Big Data Analytics

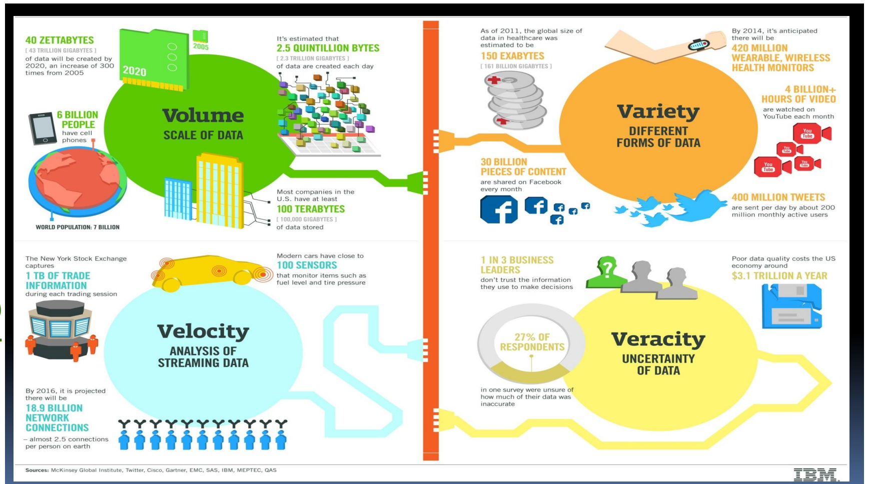
More than A CD per day?





KB -> MB -> GB -> TB -> PB -> XB -> ZB -> YB -> BB -> JB




40 ZETTABYTES
[43 TRILLION GIGABYTES]
of data will be created by 2020, an increase of 300 times from 2005
2020
2005
It's estimated that
2.5 QUINTILLION BYTES
[2.3 TRILLION GIGABYTES]
of data are created each day
6 BILLION PEOPLE have cell phones
Volume
SCALE OF DATA
WORLD POPULATION: 7 BILLION
Most companies in the U.S. have at least
100 TERABYTES
[100,000 GIGABYTES]
of data stored
As of 2011, the global size of data in healthcare was estimated to be
150 EXABYTES
[161 BILLION GIGABYTES]
By 2014, it's anticipated there will be
420 MILLION WEARABLE, WIRELESS HEALTH MONITORS
Variety
DIFFERENT FORMS OF DATA
4 BILLION+ HOURS OF VIDEO
are watched on YouTube each month
30 BILLION PIECES OF CONTENT
are shared on Facebook every month
400 MILLION TWEETS
are sent per day by about 200 million monthly active users
The New York Stock Exchange captures
1 TB OF TRADE INFORMATION
during each trading session
Modern cars have close to
100 SENSORS
that monitor items such as fuel level and tire pressure
Velocity
ANALYSIS OF STREAMING DATA
By 2016, it is projected there will be
18.9 BILLION NETWORK CONNECTIONS
– almost 2.5 connections per person on earth
1 IN 3 BUSINESS LEADERS
don't trust the information they use to make decisions
Poor data quality costs the US economy around
$3.1 TRILLION A YEAR
27% OF RESPONDENTS
in one survey were unsure of how much of their data was inaccurate
Veracity
UNCERTAINTY OF DATA
Sources: McKinsey Global Institute, Twitter, Cisco, Gartner, EMC, SAS, IBM, MEPTEC, QAS
IBM.

IoT & Big Data

- Home Automation
  - ON/OFF appliance
  - Report state of appliance periodically.
- Huge Storage
  - High velocity, High volume & High variety.
- Process data
  - Analyze Time/Day of max/min usage.
  - Decide state of appliance - minimize electricity.
  - Invent new products, new marketing schemes.

Data Engg vs Data Science

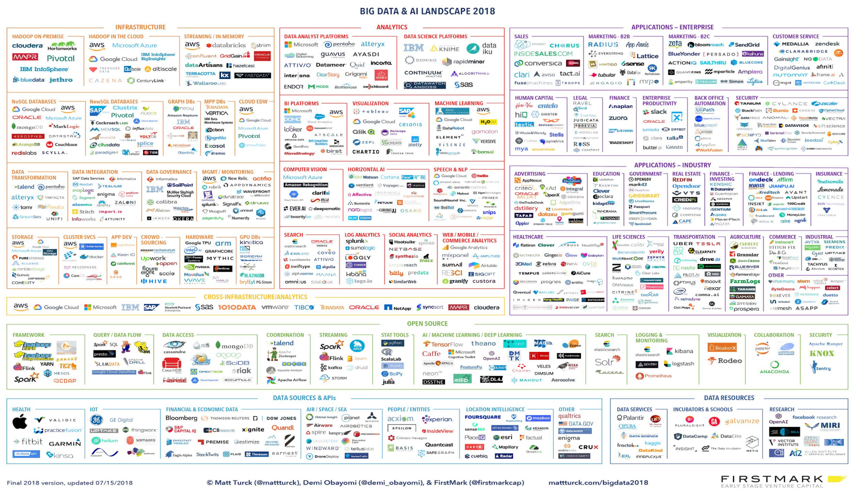
- Databases
  - RDBMS & NoSQL
  - Data Warehousing
- Big Data
  - Hadoop, Spark, HDInsight, BigTable, ...
- Infrastructure
  - Linux, Cloud Computing
- Parallel computing
- Data Science
  - Statistics and Machine learning
  - R Programming, Python
- Data Analytics
  - Data Visualizations & Business Decisions
  - Tableau, QlikView, ...

## Big Data

- It is all about :- Think, Collect, Manage, Analyze, Summarize, Visualize, Discover Knowledge and Take Decisions.
- Domains: Health-care, Retails, Trading/Share market, Finance, Security, Fraud, Search engines, Log Analysis, Telecom, Traffic Control, Manufacturing and lot more.

Scope for Big Data

- By 2020, more than 60 billion dollar industry.
- Job profiles:
  - Big Data Developer, IT operations
  - Data analyst, Data Scientist, Business Intelligence
  - Big Data Architect
- The sexiest job in the 21st century require a mixture of multi-disciplinary abilities and suitable candidates must be prepared to learn and develop constantly.

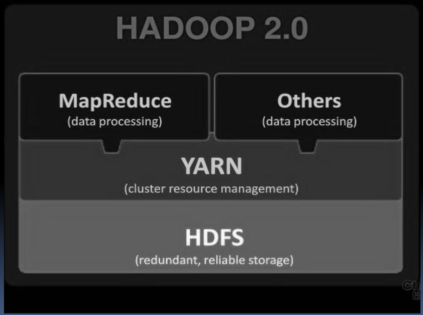

BIG DATA & AI LANDSCAPE 2018
INFRASTRUCTURE
ANALYTICS
APPLICATIONS – ENTERPRISE
APPLICATIONS – INDUSTRY
CROSS-INFRASTRUCTURE/ANALYTICS
OPEN SOURCE
DATA SOURCES & APIs
DATA RESOURCES
Final 2018 version, updated 07/15/2018
© Matt Turck (@mattturck), Demi Obayomi (@demi_obayomi), & FirstMark (@firstmarkcap)
mattturck.com/bigdata2018
FIRSTMARK
EARLY STAGE VENTURE CAPITAL

Sunbeam Infotech

Apache Hadoop

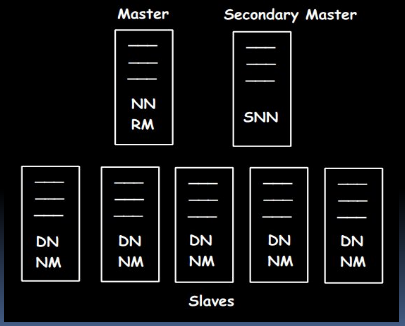
Hadoop 2.x

- Open source implementation of Big Data.
  - Doug Cutting
- Apache Software Foundation
  - Hadoop - Core/Kernel/Platform



Hadoop 2.x Daemons

- Hadoop framework is implemented in Java.
- It is made up of few java processes running in background.
- HDFS Daemons
  - Name Node, Secondary Name Node, Data Node (s)
- MapReduce/YARN Daemons
  - Resource Manager, Node Manager (s)



Hadoop Eco Systems

Hadoop Distributions

- Installing, monitoring, troubleshooting and maintaining Hadoop cluster along with Eco-systems is tedious.
- Hadoop distributions provide installations & tech-support for Hadoop eco-systems commercially.
- Some of the services are available on-premise and most of them are cloud based.
- Most popular are: Cloudera, HortonWorks, MapR, IBM, Intel, Microsoft, AWS and Pivotal Software.
- These services are preferred in production due to their stability, reliability and cost effectiveness.

Nilesh Ghule <nilesh@sunbeaminfo.com>
Technical Head, Sunbeam Infotech.

SUNBEAM

THANK YOU!