Database Management System

Database - collection of data stored in a format that is easily accessible.
We use DBMS to manage data.

RDBMS- (Relational DB)

- Data stored in tabular form related to each other.
- SQL structured query language is used to work with RDBMS.
- Eg. MySQL. SQL server, Oracle.

NoSQL-(Non-Relational DB)

- We don't have tables of relations.
- Don't understand SQL

SQL

- Case insensitive
- Every row is called a record.

Create and change database

CREATE DATABASE name;
USE database;

Delete database

DROP DATABASE [IF EXISTS] database_name; - IF EXISTS is optional

COMMENT: add '**--**' in front of query

Browsing:

```
SHOW DATABASES;  -Display all databases
SHOW TABLES;  -Display all tables
SHOW FIELDS FROM table / DESCRIBE table;  -Describe a table
SHOW CREATE TABLE table;  -Create table
SHOW PROCESSLIST;
KILL process_number;
```

CREATE TABLE:

The CREATE TABLE statement allows you to create a new table in a database.

`CREATE TABLE table (field1 type1, field2 type2);` -Type is datatype
`CREATE TABLE table (field1 type1, field2 type2, INDEX (field));`

-**Creating Primary Key**

- It is a unique for each record
- Only 1 in a table.
- Uniquely identifies tuples(rows) in that table
- It can not be NULL

```
CREATE TABLE table (field1 type1 PRIMARY KEY, field2 type2);
CREATE TABLE table (field1 type1, field2 type2, PRIMARY KEY (field1));
CREATE TABLE table (field1 type1, field2 type2, PRIMARY KEY
(field1,field2));
```

`CREATE TABLE table (field1 type1 NOT NULL, field2 type2);` -Specifying that col to not to be Null.

-**Creating Foreign Key**

- Key that points to primary key of another table
- Acts as cross reference between tables

```
CREATE TABLE table1 (fk_field1 type1, field2 type2, ...,
  FOREIGN KEY (fk_field1) REFERENCES table2 (t2_fieldA))
    [ON UPDATE|ON DELETE] [CASCADE|SET NULL]
```
-optional

```
CREATE TABLE table1 (fk_field1 type1, fk_field2 type2, ...,
 FOREIGN KEY (fk_field1, fk_field2) REFERENCES table2 (t2_fieldA,
t2_fieldB))
```

-**Creating table if not exists**

This will check the table name, and if not exist, then will create a table.
If it exists, it will not create a new table.

```
CREATE TABLE table IF NOT EXISTS;
```

<u>DROP TABLE</u>:

`DROP TABLE table;` -will delete table
`DROP TABLE IF EXISTS table;` -checks existence and deletes table
`DROP TABLE table1, table2, …` -deletes multiple table

INSERT:
Inserting data into table

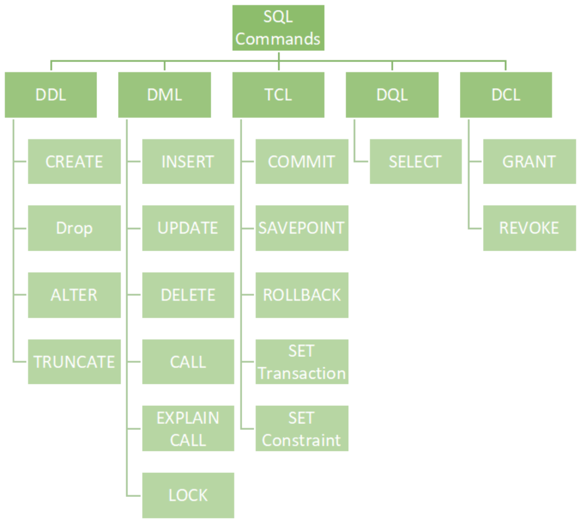
```
INSERT INTO table_name VALUES(val1, val2,..);
INSERT INTO table_name(col1, col2,..) VALUES(val1, val2,..);
INSERT INTO table_name(col1, col2,..) VALUES(a1, a2,..),(b1, b2,..),
(c1,c2,..);
```
–inserting multiple values (separated by commas)

DELETE:
Deletes the row (table data)

`DELETE * FROM table1` –By default deletes all rows → DML
`TRUNCATE TABLE table1` –Deletes all data of table → DDL
`DROP TABLE table1` –Delete the entire table and its content → DDL
`DELETE FROM table1 WHERE condition` –Deletes the rows specified using WHERE
`DELETE FROM table1, table2 WHERE table1.id1 = table2.id2 AND condition` –Deletes the rows based on condition

Delete vs Truncate

DELETE (DML)	TRUNCATE (DDL)
Specific row(s) can be deleted.	Deletes all the rows from the table.
Where clause can be used.	No use of where clause.
Rollback is possible	We cannot roll back.
Slower than truncate	Faster than delete.



ALTER TABLE:

-Modify

`ALTER TABLE table` **`MODIFY`** `col1 type1` -Modify column type

`ALTER TABLE table` **`MODIFY`** `col1 type1 NOT NULL ...` -Modify column type and null value

-Change

`ALTER TABLE table` **`CHANGE`** `old_col_name1 new_col_name1 type1` -Change column name

`ALTER TABLE table` **`CHANGE`** `old_name_field1 new_name_field1 type1 NOT NULL ...` -Change column name and null value

-Rename table name

`ALTER TABLE old_table_name` **`RENAME TO`** `new_table_name`

```
ALTER TABLE table ALTER field1 SET DEFAULT ...
ALTER TABLE table ALTER field1 DROP DEFAULT
```

`ALTER TABLE table` **`ADD`** `col_name1 type1;` -Adding new column

`ALTER TABLE table` **`ADD`** `col_name1 type1,` **`ADD`** `col_name2 type2;` -Adding multiple new columns

`ALTER TABLE table` **`ADD`** `col_name type1` **`FIRST`**`;` -Adding new column as FIRST column

`ALTER TABLE table` **`ADD`** `col_name type1` **`AFTER`** `another_field;` -Adding new column AFTER existing specified column

-Drop column

`ALTER TABLE table` **`DROP`** `col_name;`

-Add index

`ALTER TABLE table` **`ADD INDEX`** `(field);`

-Adding Primary Key using ALTER TABLE

`ALTER TABLE table` **`ADD PRIMARY KEY(col(S))`**`;`

-Adding Foreign Key using ALTER TABLE

`ALTER TABLE table` **`ADD FOREIGN KEY foreign_keyname (col(S))`** `REFERENCE parent_tablename (col(s));`

→ adding forieign key constraint

```
ALTER TABLE Orders
ADD CONSTRAINT FK_PersonOrder
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

-Drop Foreign Key using ALTER TABLE

`ALTER TABLE table DROP FOREIGN KEY fk_name;`

`ALTER TABLE Orders DROP CONSTRAINT FK_PersonOrder;`

SELECT:

`SELECT * FROM table;` -Display all data of that table
`SELECT * FROM table1, table2;`

`SELECT field1, field2 FROM table1;` -Display all data of columns field1 and 2 of that table
`SELECT field1, field1+10 FROM table1;` -Adds a new column with column name 'field1+10' of addition displaying column of added values
`SELECT field1, field1+10 AS 'new col' FROM table1;` -Adds a new column with column name 'new col' of addition displaying column of added values (new_col is alias)
`SELECT field1, field2 FROM table1, table2;`
`SELECT DISTINCT field1 FROM table;` -DISTINCT keyword displays unique data of column field1 of that table

- **SELECT** with **WHERE**

`SELECT * FROM table1 WHERE condition;` -Display all data of that table wrt that condition
`SELECT ... FROM ... WHERE condition;` -Display data wrt that condition

Operators : >, <, >=, <=, =, != or <> (both are not equality)

UPDATE:

The UPDATE statement updates data in a table. It allows you to change the values in one or more columns of a single row or multiple rows.

`UPDATE tablename SET col_name=new_value WHERE condition;` -Updates data with new_value of the column col_name of table tablename.
WHERE is to specify condition (for a specific row(s)), its optional, if not used WHERE, it will affect all the rows
`UPDATE table1 SET col1=new_value1, col2=new_value2, ... WHERE condition;`
-Updates multiple column elements

```
UPDATE table1, table2 SET field1=new_value1, field2=new_value2, ... WHERE
  table1.id1 = table2.id2 AND condition;
```

1. AND- if all conditions are True

```
SELECT col1, col2,... or (*) FROM table_name
WHERE condition1 AND condition2 AND ... ;
```

2. OR- if any one of the condition is True

```
SELECT col1, col2,... or (*) FROM table_name
WHERE condition1 OR condition2 OR ... ;
```

3. NOT - negates the condition

```
SELECT col1, col2,... or (*) FROM table_name
WHERE NOT condition;
```

4. IN

- shorthand for OR, allows multiple values to check in WHERE condition.
- determine if a value matches any value in a list of values.
- basically combination of multiple OR

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

5. BETWEEN

- The BETWEEN operator is a logical operator that specifies whether a value is in a range or not.
- begin and end values are included.
- values can be numbers, text, or dates

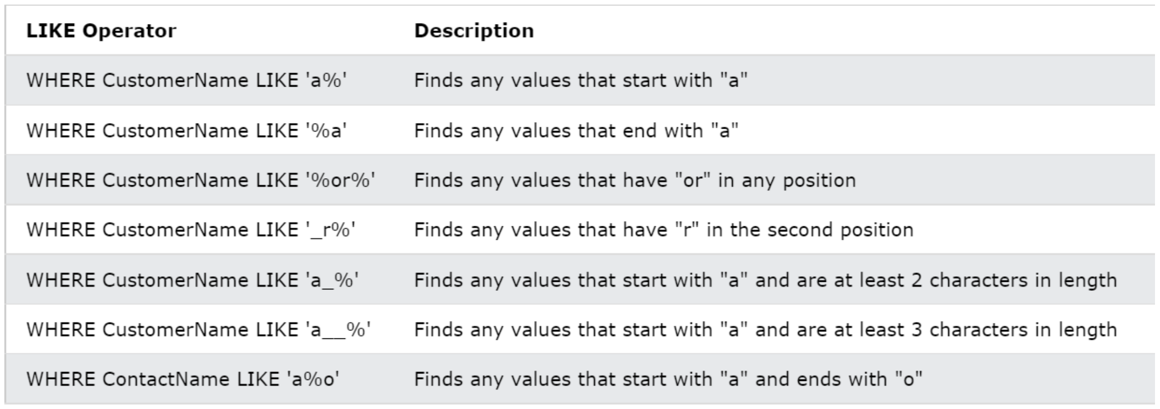
```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

6. LIKE

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

%	- for 0, 1 or multiple characters
_	- for1 single character

```
SELECT column1, column2, ...
FROM table_name
WHERE columnN LIKE pattern;
```



LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that start with "a"
WHERE CustomerName LIKE '%a'	Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%'	Finds any values that start with "a" and are at least 2 characters in length
WHERE CustomerName LIKE 'a__%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

7. EXISTS
The EXISTS operator is often used to test for the existence of rows returned by the subquery.

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS
(SELECT column_name FROM table_name WHERE condition);
```

NOT EXISTS

```
SELECT column_name(s)
FROM table_name
WHERE NOT EXISTS(SUBQUERY);
```

With ANY and ALL operators, we can perform a comparison between a single column value and a range of other values.

The operator must be a standard comparison operator (=, <>, !=, >, >=, <, or <=)

8. ANY
- means that the condition will be true if the operation is true for **any** of the values in the range.

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY(subquery);
```

9. ALL
- ALL means that the condition will be true only if the operation is true for **all** values in the range.
- used with SELECT, WHERE and HAVING statements

```
SELECT column_name(s)
```

```
FROM table_name
WHERE column_name operator ALL(SUBQUERY);
```

10. UNION

Functions:

1. COUNT()

- Returns no of rows (all or the ones that match the condition)

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

`COUNT(*)` – Returns count of everything including duplicates, null and not null
`COUNT(expression)` – Returns not null
`COUNT(DISTINCT expression)` – Returns distinct not null

2. AVG() - returns average of values of “numeric” column

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

3. SUM() - returns sum of values of “numeric” column

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

4. MIN() - The MIN() function returns the smallest value of the selected column.

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

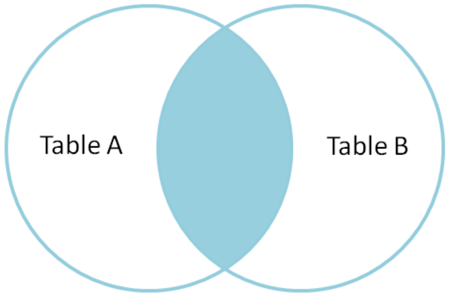
5. MAX() - The MAX() function returns the largest value of the selected column.

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

JOINS

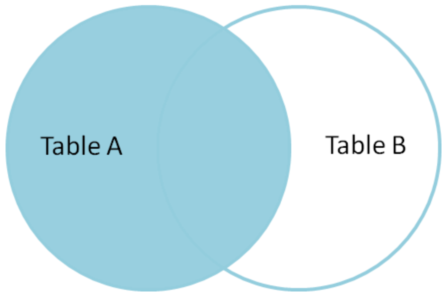
1. **INNER JOIN:**
    - Selects common values from both tables according to the condition
    - JOIN = INNER JOIN → default

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
INNER JOIN or JOIN table2
ON table1.matching_column = table2.matching_column;
```



2. **LEFT JOIN:**
    - This join returns **all** the rows of the table on the **left** side of the **join** and **matches** rows for the table on the **right** side of the join.
    - For the rows with no matching row on the right side, will give ***null***.
    - LEFT JOIN is also known as LEFT OUTER JOIN.

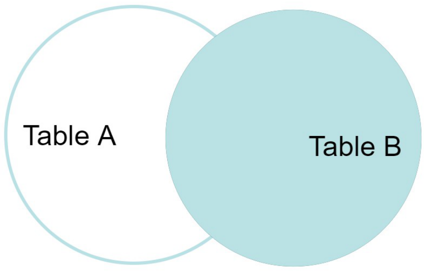
```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
LEFT JOIN table2
ON table1.matching_column = table2.matching_column;
```



3. **RIGHT JOIN:**
    - This join returns **all** the rows of the table on the **right** side of the **join** and **matches** rows for the table on the **left** side of the join.
    - For the rows with no matching row on the left side, will give ***null***.

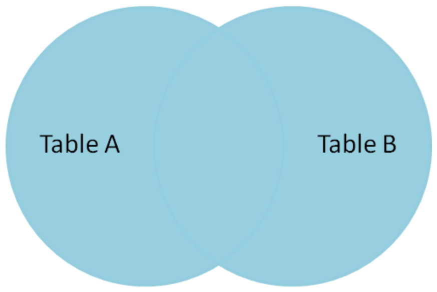
- RIGHT JOIN is also known as RIGHT OUTER JOIN.

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;
```



4. **FULL JOIN:**
   - **combining** results of both LEFT JOIN and RIGHT JOIN.
   - The result-set will contain **all** the rows from both tables.
   - For the rows for which there is **no matching**, the result-set will contain ***NULL*** values.

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
FULL JOIN table2
ON table1.matching_column = table2.matching_column;
```



<u>**Stored Procedure:**</u>

- The stored procedure is a prepared sql query with a group of statements that can be stored and called whenever needed (like functions in python).
- These can be invoked by:
    - CALL <procedure name>
    - Triggers
    - Other stored procedures (nested functions)
    - Applications like java, python, php
- These can be reused.

Advantage:

1. Reusability
2. Performance will increase
3. Performance will be consistent

→ ***** stored function returns only 1 value, but functions can be used with select query.***

<u>**Basics**</u>

1. To create procedure:
   delimiter __
   CREATE PROCEDURE <procedure_name>
   BEGIN
   ……
   END delimiter
2. To call the procedure:
   CALL <procedure name> delimiter
3. To delete procedure:
   DROP PROCEDURE <procedure_name>;

```
Delimiter $ →If $ is used as a delimiter, then the procedure should end and be called with $ itself.
CREATE PROCEDURE procedure_name()
BEGIN
      SELECT whatever from table_name; - semicolon imp
END $
CALL procedure_name$
```

→ To use the default delimiter again to call the procedure, redefine it.

```
Delimiter ;
CALL procedure_name;
```

IN → for input → (IN variable_name datatype)
Inside BEGIN, **column_name= variable_name -** to assign user defined variable

OUT → for output → (OUT variable_name datatype)
Inside BEGIN, …. **INTO variable name -** to store output INTO user defined variable name

→stored procedure with parameters.

```
Delimiter @@
CREATE PROCEDURE getmemberinfo(IN mid varchar(10))
BEGIN

      SELECT * FROM member WHERE memberid=mid;
```

→**mid** is a user defined variable to pass as parameter and it is **assigned to memberid** which is the column of the member table.

```
END @@
Delimiter ;
CALL getmemberinfo('M003');
```

→passing parameter

<u>Triggers:</u>

- For data validation
- Special stored procedure because it can't be called using CALL, it is called automatically when data modification is done. i.e. Every time you add the data, it will get activated.
- Insert (BEFORE, AFTER)
- Update (BEFORE, AFTER)
- Delete (BEFORE, AFTER)

- Resetting the value
  like
  if numbers entered is < 0, it should reset to 0
  if numbers entered is > 0, it should reset to 100

→ **All dml operations will create new and old table**

- Old one has the original unchanged data.
- New one has the updated, deleted, inserted data.

Syntax:

```
Delimiter //
CREATE TRIGGER trigger_name BEFORE INSERT/UPDATE/DELETE ON table_name
FOR EACH ROW
BEGIN
     —----
END //
Delimiter ;
```

→To drop trigger: `DROP TRIGGER schema.trigger_name;`