








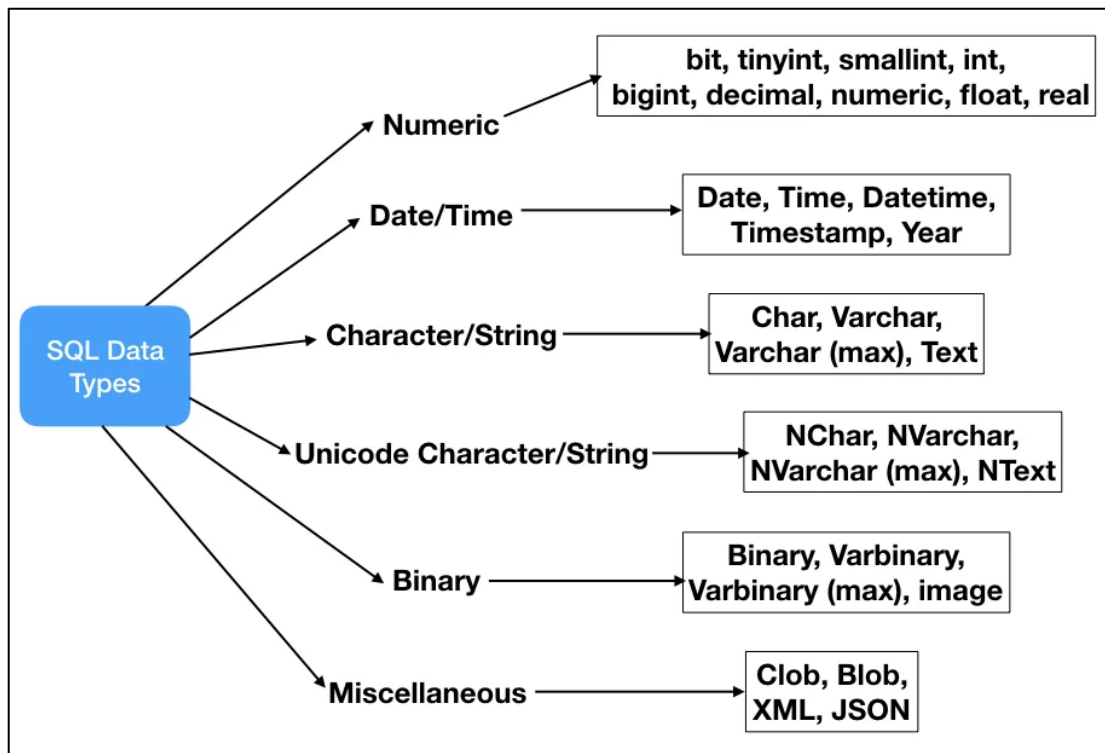
Colour Codes	
	Database
	Keywords
	Table Names
	Column Names
	Values
	Strings
	Alias (Temporary Name)

- **USE, SHOW, DESCRIBE**

1. **USE** imdb;
2. **SHOW TABLES**; *Returns table list in DB*
3. **DESCRIBE** movies;

Field	Type	Allow Null	Key	Default	Extra
id	int (11)	NO	PRI	0	
name	varchar (100)	YES	MUL	NULL	
year	int (11)	YES		NULL	
rankscore	float	YES		NULL	

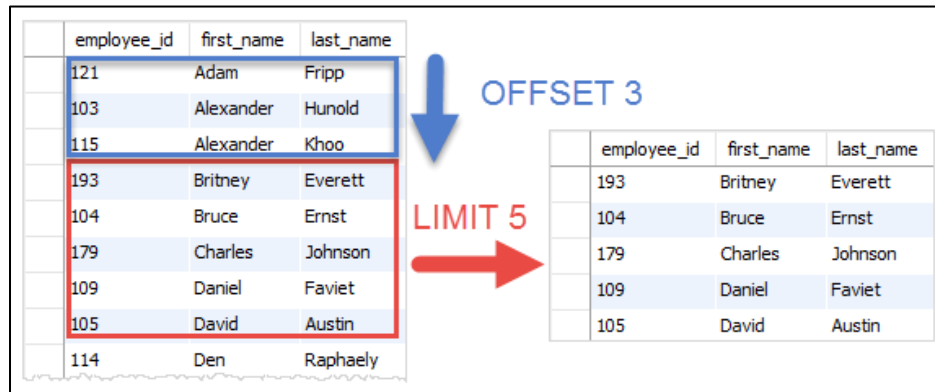
4. In any case, there are three possible values for the “Key” attribute:
  - a) If Key is empty, the column either is not indexed or is indexed only as a secondary column in a multiple-column, nonunique index.
  - b) If Key is PRI, the column is a PRIMARY KEY or is one of the columns in a multiple-column PRIMARY KEY.
  - c) If Key is UNI, the column is the first column of a UNIQUE index. (A UNIQUE index permits multiple NULL Values, but you can tell whether the column permits NULL by checking the Null field.)
  - d) If Key is MUL, the column is the first column of a nonunique index in which multiple occurrences of a given value are permitted within the column.
5. [Data Types](#)



- **SELECT**

1. `SELECT * FROM movies;`  
{SELECT: Keyword}  
\*: All Columns, (specific column names can be specified)  
FROM: Keyword,  
movies: Table Name i.e., select all columns from movies table}
2. `SELECT name, year FROM movies;` i.e., select rows consisting of name and year columns from movies.
3. `SELECT rankscore, name FROM movies;` i.e., select rows consisting of rankscore and name columns from movies.

- **LIMIT, OFFSET:**



1. `SELECT name, rankscore FROM movies LIMIT 20;` i.e., select 20 rows consisting of name and rankscore columns from movies.
2. `SELECT name, rankscore FROM movies LIMIT 20 OFFSET 20;` i.e., select 20 rows consisting of name and rankscore columns from movies table skipping first 20 rows(offset)

- **ORDER BY:**

1. list recent 10 movies  
`SELECT name, year, rankscore FROM movies ORDER BY year DESC LIMIT 10;`
2. If **ORDER BY** not specified default: ASC  
`SELECT name, year, rankscore FROM movies ORDER BY year LIMIT 10;`

Note: the output row order may not be same as the one in the table due to query optimizer and internal data-structures/indices.

- **DISTINCT:**

1. list all genres  
`SELECT DISTINCT genre FROM movies_genres;`
2. multiple-column DISTINCT  
`SELECT DISTINCT first_name, last_name FROM directors;`

- **WHERE**(Conditional outputs: **TRUE, FALSE, NULL**):

1. list all movies with rankscore > 9;  
`SELECT name, year, rankscore FROM movies WHERE rankscore > 9;`  
`SELECT name, year, rankscore FROM movies WHERE rankscore > 9 ORDER BY rankscore DESC LIMIT 20;`
2. Comparison Operators: =, <>, !=, <, <=, >, >=

Equal to	=
Not equal to	<> or !=
Greater than	>
Less than	<
Greater than or equal to	>=
Less than or equal to	<=

3. `SELECT * FROM movies_genres WHERE genre = 'Comedy';`  
`SELECT * FROM movies_genres WHERE genre <> 'Horror';`  
`NULL => does not-exist/unknown/missing`
4. `"="` does not work with `NULL`, will give you an empty result-set.  
`SELECT name, year, rankscore FROM movies WHERE rankscore = NULL;`  
`SELECT name, year, rankscore FROM movies WHERE rankscore IS NULL LIMIT 20;`  
`SELECT name, year, rankscore FROM movies WHERE rankscore IS NOT NULL LIMIT 20;`
5. LOGICAL OPERATORS: `AND`, `OR`, `NOT`, `ALL`, `ANY`, `BETWEEN`, `EXISTS`, `IN`, `LIKE`
  - a. `AND` allows you to select only rows that satisfy two conditions.
  - b. `OR` allows you to select rows that satisfy either of two conditions.
  - c. `NOT` allows you to select rows that do not match a certain condition.
  - d. `ALL` operator returns `TRUE` iff all of the subquery's values meet the condition.
  - e. `ANY` return true if any of the subquery's values meet the condition.
  - f. `BETWEEN` allows you to select only rows within a certain range.
  - g. `IN` allows you to specify a list of values you'd like to include.
  - h. `LIKE` allows you to match similar values, instead of exact values.
  - i. `IS NULL` allows you to select rows that contain no data in a given column.

6. Website search filters

```
SELECT name, year, rankscore FROM movies WHERE rankscore > 9 AND year > 2000;
SELECT name, year, rankscore FROM movies WHERE NOT year <= 2000 LIMIT 20;
SELECT name, year, rankscore FROM movies WHERE rankscore > 9 OR year > 2007;
```

# will discuss about `ANY` and `ALL` when we discuss sub-queries

```
SELECT name, year, rankscore FROM movies WHERE year BETWEEN 1999 AND 2000;
```

#inclusive: `year >= 1999` and `year <= 2000`

```
SELECT name, year, rankscore FROM movies WHERE year BETWEEN 2000 AND 1999;
```

#low value `<=` high value else you will get an empty result set

```
SELECT director_id, genre FROM directors_genres WHERE genre IN('Comedy', 'Horror');
```

# same as `genre = 'Comedy' OR genre = 'Horror'`

```
SELECT name, year, rankscore FROM movies WHERE name LIKE 'Tis%';
```

# `%` is a wildcard character to imply zero or more characters

#`first_name` starting in `'Tis'`

```
SELECT first_name, last_name FROM actors WHERE first_name LIKE '%es';
```

#`first_name` ending in `'es'`

```
SELECT first_name, last_name FROM actors WHERE first_name LIKE '%es%';
```

#`first_name` contains `'es'`

```
SELECT first_name, last_name FROM actors WHERE first_name LIKE 'Agn_s';
```

#`'_'` implies exactly one character.

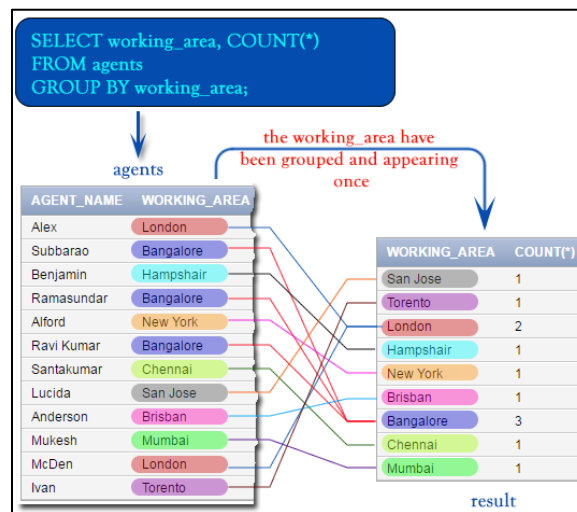
# If we want to match `%` or `_`, we should use the backslash as the escape character: `\%` and `\_`

```
SELECT first_name, last_name FROM actors WHERE first_name LIKE 'L%' AND first_name NOT LIKE 'Li%';
```

• **AGGREGATE FUNCTIONS:**

1. Computes a single value on a set of rows and returns the aggregate `COUNT`, `MIN`, `MAX`, `SUM`, `AVG`  
`SELECT MIN(year) FROM movies;`  
`SELECT MAX(year) FROM movies;`  
`SELECT COUNT(*) FROM movies;`  
`SELECT COUNT(*) FROM movies WHERE year > 2000;`  
`SELECT COUNT(year) FROM movies;`

- GROUP-BY



- find number of **movies** released per **year**

```
SELECT year, COUNT(year) FROM movies GROUP BY year;
```

```
SELECT year, COUNT(year) FROM movies GROUP BY year ORDER BY year;
```

```
SELECT year, COUNT(year) year_count FROM movies GROUP BY year ORDER BY year_count;
```

# **year\_count** is an **alias**(temporary name).

# often used with **COUNT**, **MIN**, **MAX**, or **SUM**.

# if grouping columns contain **NULL** values, all null values are grouped together.

- HAVING:

- Print **years** which have >1000 **movies** in our DB

```
SELECT year, COUNT(year) year_count FROM movies GROUP BY year HAVING year_count > 1000;
```

# specify a condition on groups using **HAVING**.

- Order of execution:

- GROUP BY** to create groups
- apply the **AGGREGATE FUNCTION**
- Apply **HAVING** condition.

- Often used along with **GROUP BY**. Not Mandatory.

```
SELECT name, year FROM movies HAVING year > 2000;
```

- HAVING** without **GROUP BY** is same as **WHERE**

```
SELECT year, COUNT(year) year_count FROM movies WHERE rankscore > 9 GROUP BY year HAVING year_count > 20;
```

- HAVING** vs **WHERE**

- WHERE** is applied on individual rows while **HAVING** is applied on groups.
- HAVING** is applied after grouping while **WHERE** is used before grouping.
- The difference between the **HAVING** and **WHERE** clause in SQL is that the **WHERE** clause cannot be used with **AGGREGATES**, but the **HAVING** clause can.

- **ORDER of Keywords (SF JOW GHOL)**

```

SELECT
  [ALL | DISTINCT | DISTINCTROW ]
  [HIGH_PRIORITY]
  [STRAIGHT_JOIN]
  [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
  [SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
  select_expr [, select_expr] ...
  [into_option]
  [FROM table_references
  [PARTITION partition_list]]
  [WHERE where_condition]
  [GROUP BY {col_name | expr | position},... [WITH ROLLUP]]
  [HAVING where_condition]
  [WINDOW window_name AS (window_spec)
  [, window_name AS (window_spec)] ...]
  [ORDER BY {col_name | expr | position}
  [ASC | DESC],... [WITH ROLLUP]]
  [LIMIT [{offset,} row_count | row_count OFFSET offset}]
  [into_option]
  [FOR {UPDATE | SHARE}
  [OF tbl_name [, tbl_name] ...]
  [NOWAIT | SKIP LOCKED]
  [LOCK IN SHARE MODE]
  [into_option]

```

```

into_option: {
  INTO OUTFILE 'file_name'
  [CHARACTER SET charset_name]
  export_options
  | INTO DUMPFILE 'file_name'
  | INTO var_name [, var_name] ...
}

```

- SELECT- \* - FROM - JOIN - ON - WHERE - GROUP BY - HAVING – ORDER BY – LIMIT
- SOME FRENCH JOKERS ON WHEELS GROUPED HAVING ORCHID LILLIES
- \*: ALL (remember even DISTINCT can be used in this place)
- G: GROUP BY
- H: HAVING
- O: ORDER BY (has ASC and DESC)
- L: LIMIT (has OFFSET)

- **JOINS:**

1. A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

# For each movie, print name and the genres

```
SELECT m.name, g.genre FROM movies m JOIN movies_genres g ON m.id = g.movie_id LIMIT 20;
```

# table aliases: m and g

2. Natural join: a join where we have the same column-names across two tables.

#T1: C1, C2

#T2: C1, C3, C4

```
SELECT * FROM T1 JOIN T2;
```

```
SELECT * FROM T1 JOIN T2 USING(C1);
```

# returns C1,C2,C3,C4

# no need to use the keyword "ON"

3. **INNER JOIN(JOIN)**: The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e., value of the common field will be same.

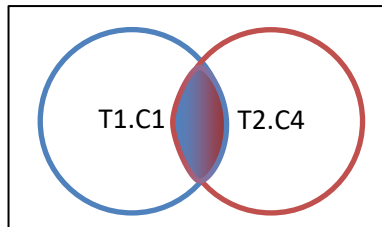
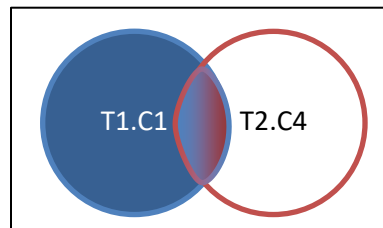


Table1(Left)			Table2(Right)		
C1	C2	C3	C4	C5	C6
1	a	b	1	0	1
2	c	d	2	1	0
4	e	f	3	0	1
5	g	h	5	1	0

`SELECT * FROM T1 JOIN T2 ON T1.C1 = T2.C4;`

Output Table					
C1	C2	C3	C4	C5	C6
1	a	b	1	0	1
2	c	d	2	1	0
5	g	h	5	1	0

4. **LEFT JOIN(LEFT OUTER JOIN)**: This join returns all the rows of the table on the LEFT side of the join and matching rows for the table on the right side of join. The rows for which there is no matching row on right side, the result-set will contain **NULL**.

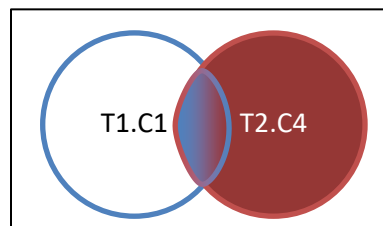


`SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.C1 = T2.C4;`

Output Table					
C1	C2	C3	C4	C5	C6
1	a	b	1	0	1
2	c	d	2	1	0
4	e	f	NULL	NULL	NULL
5	g	h	5	1	0

`SELECT m.name, g.genre FROM movies m LEFT JOIN movies_genres g ON m.id = g.movie_id LIMIT 20;`

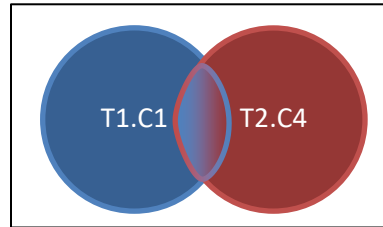
5. **RIGHT JOIN(RIGHT OUTER JOIN)**: This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join. The rows for which there is no matching row on left side, the result-set will contain null.



`SELECT * FROM T1 RIGHT OUTER JOIN T2 ON T1.C1 = T2.C4;`

Output Table					
C1	C2	C3	C4	C5	C6
1	a	b	1	0	1
2	c	d	2	1	0
5	g	h	5	1	0
NULL	NULL	NULL	3	0	1

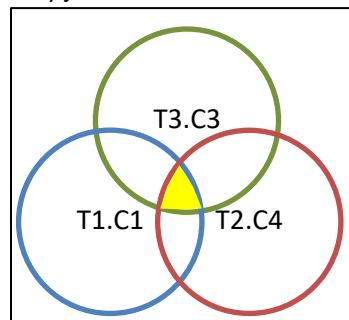
6. **FULL JOIN(FULL OUTER JOIN):** FULL JOIN creates the result-set by combining result of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both the tables. The rows for which there is no matching, the result-set will contain NULL values.



`SELECT * FROM T1 FULL JOIN T2 ON T1.C1 = T2.C4;`

Output Table					
C1	C2	C3	C4	C5	C6
1	a	b	1	0	1
2	c	d	2	1	0
4	e	f	NULL	NULL	NULL
5	g	h	5	1	0
NULL	NULL	NULL	3	0	1

7. 3-way joins and k-way joins:



`SELECT a.first_name, a.last_name FROM actors a JOIN roles r ON a.id = r.actor_id JOIN movies m ON m.id = r.movie_id AND m.name = 'Officer 444';`

Note: Joins can be computationally expensive when we have large tables.

#### • Sub-Queries(Nested Queries or Inner Queries)

1. List all **actors** in the movie **Schindler's List**

```
SELECT first_name, last_name FROM actors WHERE id IN
    (SELECT actor_id FROM roles WHERE movie_id IN
        (SELECT id FROM movies WHERE name = "Schindler's List") # or 'Schindler\'s List'
    );
```

2. Syntax:

```
SELECT column_name [, column_name ]
FROM table1 [, table2 ]
WHERE column_name OPERATOR
    (SELECT column_name [, column_name ]
    FROM table1 [, table2 ]
    [WHERE])
```

- First the inner query is executed and then the outer query is executed using the output values in the inner query
- Operators: **IN, NOT IN, EXISTS, NOT EXISTS, ANY, ALL** are Comparison operators
- **EXISTS** returns true if the subquery returns one or more records or **NULL**
- **ANY** operator returns **TRUE** if any of the subquery values meet the condition.
- **ALL** operator returns **TRUE** if all of the subquery values meet the condition.

```
SELECT * FROM movies where rankscore >= ALL(SELECT MAX(rankscore) from movies);
# get all movies whose rankscore is same as the maximum rankscore.
# e.g.: rankscore <> ALL(...)
```

3. Correlated subquery is a subquery that uses values from the outer query. Because the subquery may be evaluated once for each row processed by the outer query, it can be slow.

In following example, the objective is to find all employees whose salary is above average for their department.

```
SELECT employee_number, name
FROM employees emp
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
    WHERE department = emp.department);
```

- **WITH Clause**

1. The clause is used for defining a temporary relation such that the output of this temporary relation is available and is used by the query that is associated with the WITH clause.
2. Queries that have an associated WITH clause can also be written using nested sub-queries but doing so add more complexity to read/debug the SQL query.
3. The name assigned to the sub-query is treated as though it was an inline view or table

```
WITH <alias_name> AS (sql_subquery_statement)
SELECT column_list FROM [table_name]
[WHERE <join condition>]
```

- **Data Manipulation Language(DML): SELECT, INSERT, UPDATE, DELETE**

1. **INSERT** is used to insert data into a table. A new entry or data from other tables can be added.

```
INSERT INTO movies(id, name, year, rankscore) VALUES(412321, 'Thor', 2011, 7);
```

```
INSERT INTO movies(id, name, year, rankscore) VALUES(412321, 'Thor', 2011, 7),(412322, 'Iron Man', 2008, 7.9),
(412323, 'Iron Man 2', 2010, 7);
```

2. **UPDATE** is used to update existing data within a table. It can also Update multiple rows. It Can be used along with Sub-queries.

```
UPDATE <TableName> SET col1 = val1, col2 = val2 WHERE condition
```

```
UPDATE movies SET rankscore = 9 WHERE id = 412321;
```

3. **DELETE** is used to delete records from a database table.

```
DELETE FROM movies WHERE id = 412321;
```

- **Data Definition Language(DDL):**

1. **CREATE** – is used to create the database or its objects (like table, index, function, views, store procedure and triggers).

```
CREATE TABLE language ( id INT PRIMARY KEY, lang VARCHAR(50) NOT NULL);
```

2. **Constraints:** Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted. Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.



- a. **NOT NULL** - Ensures that a column cannot have a **NULL** value
  - b. **UNIQUE** - Ensures that all values in a column are different
  - c. **PRIMARY KEY** - A combination of a **NOT NULL** and **UNIQUE**. Uniquely identifies each row in a table
  - d. **FOREIGN KEY** - Uniquely identifies a row/record in another table
  - e. **CHECK** - Ensures that all values in a column satisfies a specific condition
  - f. **DEFAULT** - Sets a default value for a column when no value is specified
  - g. **INDEX** - Used to create and retrieve data from the database very quickly
3. **ALTER**(ADD, MODIFY, DROP) is used to alter the structure of the database.  
**ALTER TABLE language ADD country VARCHAR(50);**  
  
**ALTER TABLE language MODIFY country VARCHAR(60);**  
  
**ALTER TABLE language DROP country;**
4. **DROP** is used to delete objects from the database. It removes both the table and all of the data permanently.  
**DROP TABLE TableName;**  
**DROP TABLE TableName IF EXISTS;**
5. **TRUNCATE** is used to remove all records from a table, including all spaces allocated for the records are removed.  
**TRUNCATE TABLE TableName;**
- **Data Control Language(DCL)**
  1. DCL includes commands such as **GRANT** and **REVOKE** which mainly deal with the rights, permissions, and other controls of the database system.
    - a. **GRANT**-gives user's access privileges to the database.
    - b. **REVOKE**-withdraw user's access privileges given by using the **GRANT** command.
- **TCL(transaction Control Language):**
  1. TCL commands deal with the transaction within the database.
    - a. **COMMIT**– commits a Transaction.
    - b. **ROLLBACK**– rollbacks a transaction in case of any error occurs.
    - c. **SAVEPOINT**–sets a save point within a transaction.
    - d. **SET TRANSACTION**–specify characteristics for the transaction