

- * SQL stands for standard Query language
- * SQL is a standard way to query / obtain / add / delete / modify data
- * It is used by data scientist, machine learning engineer, S/w engineer etc
- * SQL is not a general purpose programming language. It is a domain specific language whose primary task is to build database. Other languages like C/C++/Java/Python are general purpose programming language and therefore use to build web servers, operating system, while SQL is not used for such tasks.
- * SQL is called declarative programming language where we do not specify step-by-step procedure to get / obtain data.

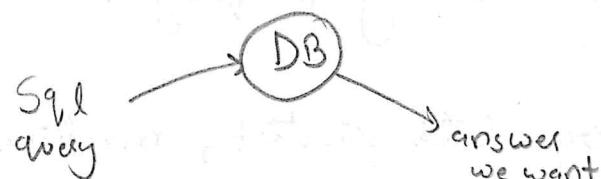
Ex- SELECT movie_name from movies WHERE Year > 200
 here, we are declaring "what we want?" and
 not, "how to get it?"

Because it is declarative, that is why it becomes simple ~~for the~~ to use.

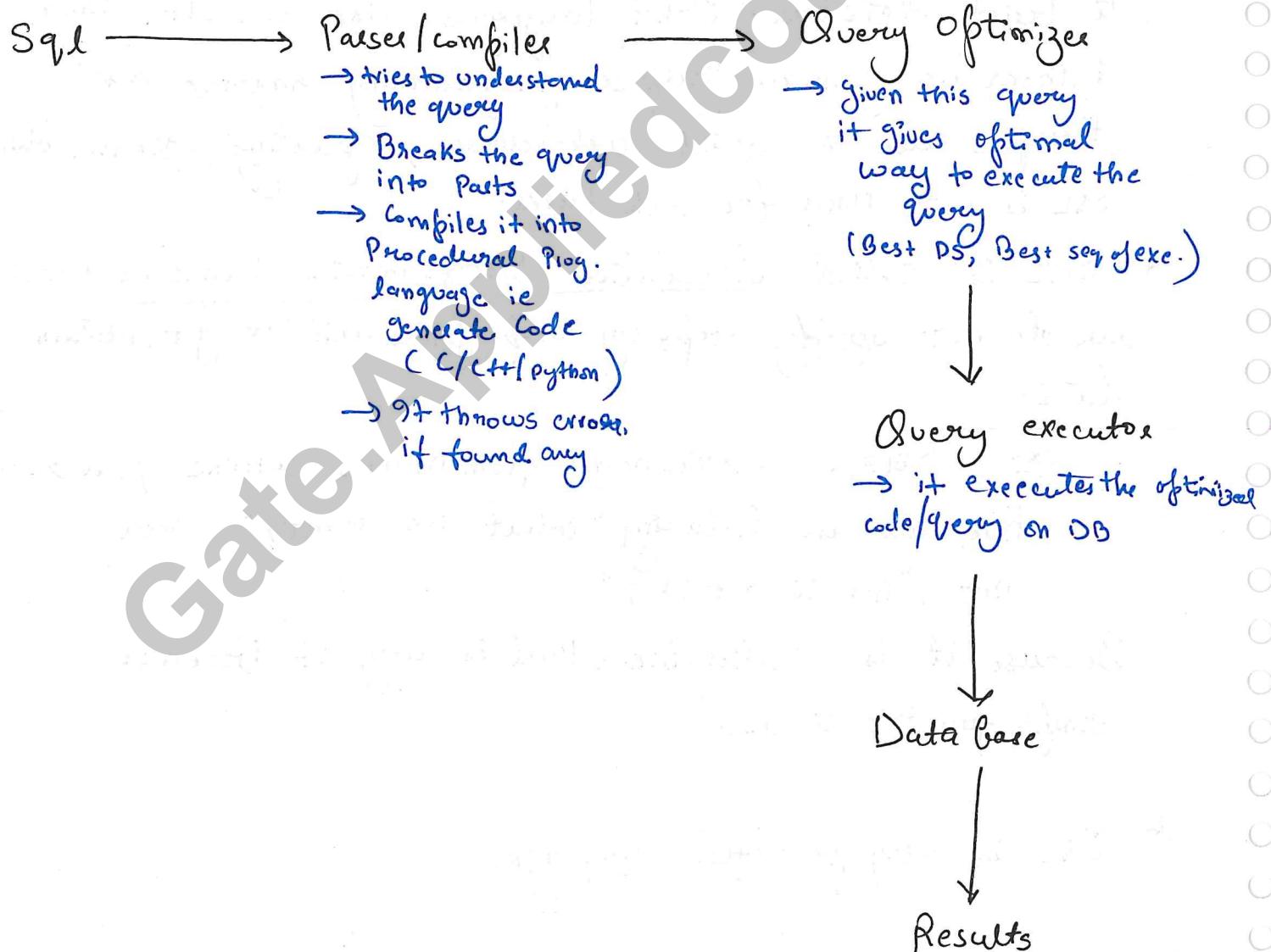
- * SQL is very powerful language.

7.3 Execution of SQL statement

Given the fact that SQL is an declarative programming language, "how does an SQL statement is executed?"

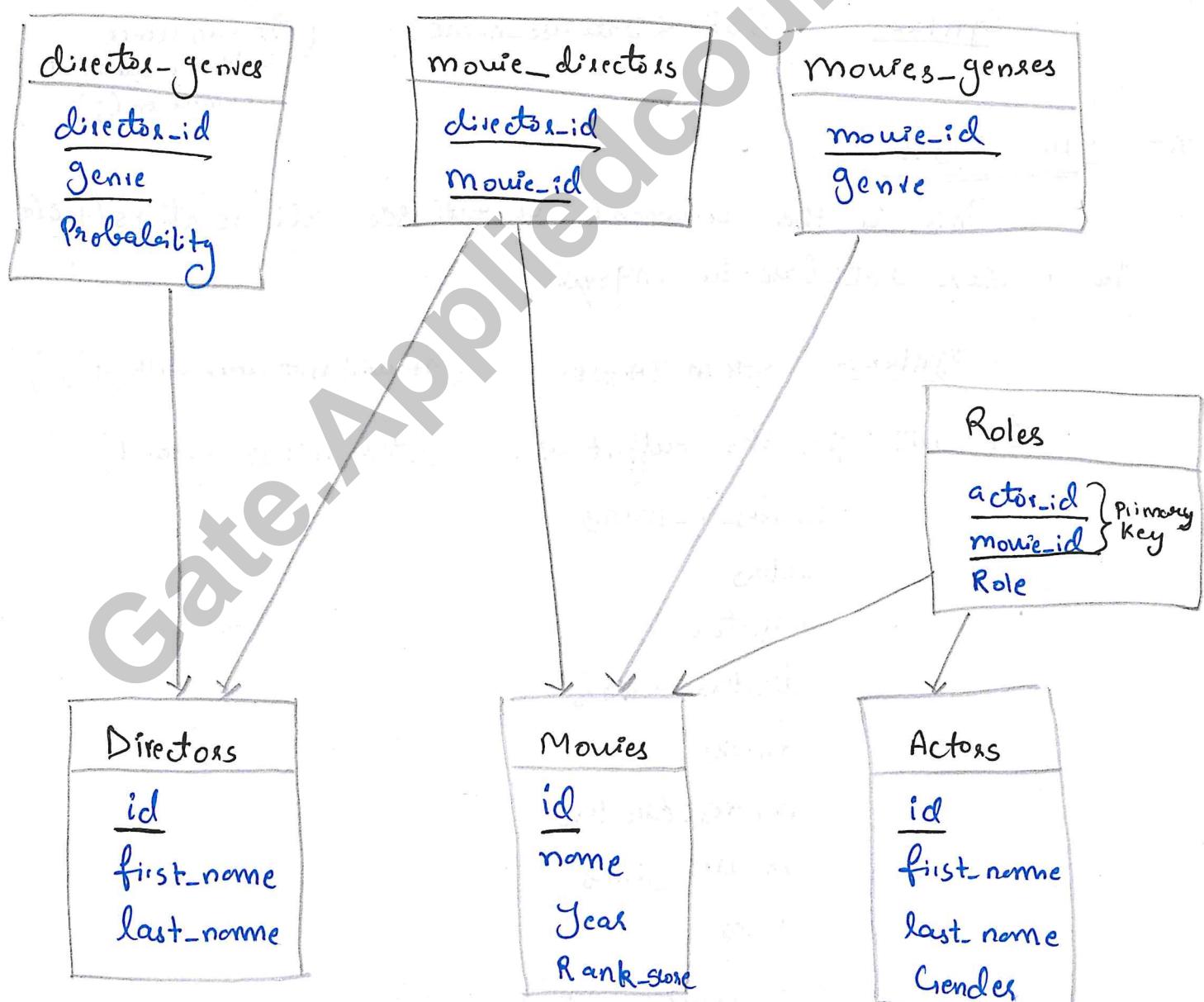


So, internally it works as follows:



- * IMDB is a website owned by Amazon. This dataset will be considered as an set example to understand different SQL commands. This dataset is a movie dataset.
- * This data set have 388,269 movies from 1888-2008 which have 817,718 actors and 86,880 directors.

→ Structure of the IMDB dataset (Relational database):



S Q L COMMANDS

P-1 Use, Describe , Show-tables:

*- USE :

It is used to tell mysql that which database we want use or work on.

Syntax -

USE <Database_name>;

(It can work with or without semicolon (;))

*- SHOW TABLES :

This is the command to see all relations/tables in a given database in mysql .

Syntax -

SHOW TABLES;

(It will not work without ';')

It will give the output as :

(for IMDB dataset)

Tables_in_IMDB
actors
directors
directors_genres
movies
movies_directors
movies_genres
roles

7 rows in set (0.00 sec)

DESCRIBE:

To understand each of this table, we use a command DESCRIBE. It will give complete info about a relation like , data type, default values, keys etc.

Syntax- DESCRIBE <Table-name>; (will not work without semicolon ;)

Example- DESCRIBE actors;

output \Rightarrow

Field	Type	Null	Key	default	Comments
id	int(11)	No	PRI	0	
first-name	Varchar(100)	YES	MUL	NULL	
last-name	Varchar(100)	YES	MUL	NULL	
Gender	Char(1)	YES		NULL	

datatype

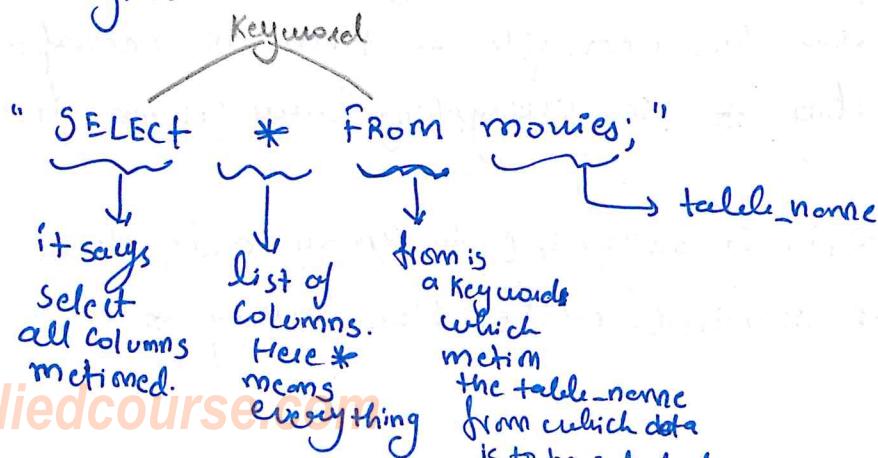
4 rows in set (0.00 sec)

it means more than 1 actor could have same first-name or last-name.
Ex- Both Salman and Shahrukh have last name 'Khan'

Q.2 Select:

Suppose we are given with a task : "list all movies in the database".

Given a query "SELECT * FROM movies;"



It will be read as "Select ~~Everything~~ (*) from the table name movies". So, as soon as we run it, it will give complete table content as the result.

NOTE - ~~Select~~ movies will only result in the description of every attribute present in table movies. While, select basically ~~says~~ says which column to select and will give all values / content present under that column.

Example 2: "SELECT name, year FROM movies;"

→ it will give 388,205 row as o/p ~~with~~ with only two attributes "name and year" from table movies.

NOTE - Here we are not explaining how to generate output, we only mention what we want.

NOTE - The output generated by "Select" in SQL query is called "result set". It is nothing but another table that it creates. (It is a set of rows with column name)

NOTE - Example 2 is faster than ~~example 1~~ example 1 (list all movies) bcoz amount of data to be transferred is less in example 2 than in example 1 (∴ it is selecting *, ie everything / every column from table movies).

So, it is suggested to query only those attributes which are required rather than using *.

NOTE: The order of column name could be anything.

It need not to be in the same order as it is given in the table/relation. The o/p or result set will have the order of the attributes as given in the query.

Note that, the order of rows in result set will be same as order in the original table/relation. This is called row-order preservation (So, row orders are preserved by simple select queries, there are some queries where row order is not preserved)

8.3 Limit, Offset:

Limit is a keyword which tells the number of rows we need to limit in our result set.

Example: SELECT name, year FROM movies LIMIT 20;

	name	Year
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		

20 rows
will be the
output.

↳ # of
rows

it will give
only first
20 rows as
outputs.

Here, we got the first 20 rows only. What if we want to get the next 20 rows now. For that we use another keyword called OFFSET.

When we add offset <value> it means "ignore the ^{starting} rows equal to offset value and print the next rows equal to limit value."

O/P \Rightarrow it will ignore first 20 rows and will print next 20 rows of table movies with columns name and year.

So, offset basically says number of rows from start to ignore. If offset is equal to 40, it will ignore first 40 rows of table movies.

Limit and offset makes the data more readable as they provide us subset of data.

Q.4 Order by:

This keyword is used to get the ordered output for a given column name.

If the order type is not mentioned, it by default order the data in ascending order (ASC)

Example 1: SELECT name, year FROM movies ORDER BY year DESC LIMIT 10;

The result is ordered based on year.
The result is in descending order.

The feature like "sort by" and "filter" on various websites run the SQL command with ORDER BY at backend.

Ph: 844-844-0102

Hence, it helps us to sort the data and therefore the order of ^{rows in} result set will not be same as the original table rows order.

Q.5 Distinct:

This keyword is used to get unique / distinct values from a column / attribute.

Example- SELECT DISTINCT genre FROM movies_genres;

↑
Keyword ↓
Column-name

It will o/p all distinct genres. The o/p set will have 21 rows for 21 different genres for the given IMDB dataset.

If we want to find distinct over multiple fields we can write it as:

Example 2: SELECT DISTINCT first_name, last_name
FROM Directors; ORDER BY first_name;

Distinct will work/applied on both of these.

It will give 86844 rows as o/p, there ~~are two~~ ^{are two} rows with first_name = "Zvonko" because there are two "Zvonko" with different last name i.e. 'Coh' and 'Saksida' respectively.

So, when distinct is applied over multiple columns then complete row should be distinct (one column value can repeat but should have different value in other columns)

8.6 Where, Comparison operators, NULL: Ph: 844-844-0102

→ WHERE:

This keyword is used to filter the result and apply conditions.

Example 1: SELECT name, year, rank-score FROM movies WHERE rank-score > 9;

↓ condition

it says generate
only those rows
which have rank-score > 9

and finally for the rows which have rank-score > 9 we will select these name, year and rank-score.
It will output 1069 rows which are stated > 9 .

Example 2: SELECT name, year, rank-score FROM movies
WHERE rank-score > 9 ~~and~~ ORDER BY rank-score DESC LIMIT 20;

→ It will give top 20 movies, where rank-score > 9 .
name, year, rank-score

NOTE: Whatever comes after WHERE clause / command is called 'Condition'.

So, a condition's output could be True, False and NULL and it may have comparison operators like $=$, $\underline{<} \neq$, $<$, \leq , $>$, \geq

not equal to

Example 3:

```
SELECT * FROM movies_genres WHERE  
genre = "Comedy";
```

⇒ It will select all columns where genre is equal to comedy. It will o/p 56425 rows.

Example 4:

```
SELECT * FROM movies_genres WHERE  
genre <> "Horror";
```

OR

```
SELECT * FROM movies_genres WHERE  
genre != "Horror";
```

⇒ It will select all columns ~~where~~ of all genres except "horror".

→ NULL:

NULL in SQL means that : Value is unknown / missing / does not exist

NULL is a special keyword in SQL. For any attribute value as NULL means that value does not exists.

NOTE

The operator "=" does not work with NULL

Example 5:

```
SELECT name, year, rankscore  
FROM movies WHERE rankscore = NULL;
```

⇒ O/p will be an empty set. We can't compare NULL with "equal to" symbol.

In order to make it work, we write the query using "IS" and "IS NOT".

Example 2 - SELECT name, year, rank-score FROM movies

WHERE rank-score IS NULL; LIMIT 20;



it will work to give all rows where rank-score is NULL

- It will op 20 rows with rank-score as NULL

Example 3 - SELECT name, year, rank-score FROM movies WHERE rank-score IS NOT NULL LIMIT 20;

will op all rows where rank-score is not having NULL value.

8.7 logical operators:

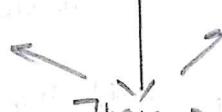
AND, OR, NOT, ALL, ANY, BETWEEN, EXISTS, IN, LIKE, SOME are some of the logical operators in SQL. Logical operators are required whenever we need to put multiple filters.

AND

Example 1 : SELECT name, year, rank-score FROM movies WHERE rank-score > 9 AND year > 2000;

Condition 1

Condition 2



These 2 Conditions are combined using AND operator.

⇒ It will list all movies that are released after year 2000 and having rank-score > 9. It will o/p 250 rows.

NOT

Example 2: SELECT name, year, rank-score FROM movies
WHERE NOT year <= 2000 LIMIT 20;

this Condition
is equivalent to
Year > 2000

⇒ It will list all movies that are released after year 2000 and limit the result set to the top 20 rows.

OR

Example 3: SELECT name, year, rank-score FROM movies
WHERE rank-score > 9 OR year > 2007;

Condition 1 Condition 2

↓
Combined Condition 1 and 2
and will result to TRUE if
any of the condition gets true.

⇒ It will list all movies that are either released after 2007 or having rank-score > 9.

NOTE - ALL and ANY are discussed in topic Subqueries.

Example 4: SELECT name, year, rank score FROM movies
(with BETWEEN) WHERE year BETWEEN 1999 AND 2000;

→ changed from 1999 - 2000
OR → low value

(without BETWEEN)
SELECT name, year, rank score FROM movies
WHERE year ≥ 1999 AND year ≤ 2000 ;

→ It will list all movies released in year between 1999 to 2000.

This is called inclusive range because both the values i.e. 1999 and 2000 are included in result.

Example 5: SELECT name, year, rank score FROM movies
WHERE year BETWEEN 2000 AND 1999

⇒ This query will not work because low value is greater than high value

NOTE

low value should be \leq high value otherwise we will get an empty set as result.

Example 6: SELECT director_id, genre FROM directors_genres
 (with IN) WHERE genre IN ('Comedy', 'Horror');

OR

(without IN) SELECT director_id, genre FROM directors_genres
 WHERE genre = 'Comedy' OR genre = 'Horror';

⇒ It will list all director_id and genre where genre is either Comedy or horror. So, IN clause allows us to test if expression matches any value present in the list or set of values.

LIKE (used for text matching)

Example 7: SELECT name, year, rank_score FROM movies
 WHERE name LIKE 'Tis%';

(here % is the wildcard character to imply zero or more ~~than~~ one character)

⇒ It will list all movies name, year, rankscore where names start with 'Tis' (ie 'Tis' is followed by anything or zero)

(In shell command 'grep' or regular expression in ToC works like this)

Example 8: SELECT first-name, last-name FROM actors WHERE first-name LIKE '%es';

⇒ This will o/p all first-name, last-name of actors whose first-name are ending with 'es'

Example 9: SELECT first-name, last-name, FROM ~~actors~~ WHERE first-name LIKE '%.es%';

⇒ This will o/p all first-name, last-name of actors whose first-name contains substring 'es'

Example 10: SELECT first-name, last-name FROM actors WHERE first-name LIKE 'Agn_s';

('_' (underline) is also a wildcard character which implies exactly one character)

⇒ This will o/p all first-name, last-name of actors whose first-name is like 'Agn_s'

↑
only 1
character
can fix
in to this.

NOTE

Consider the below relation:

T :

Name	percentage
n ₁	59%
n ₂	63%
n ₃	96%

Query: SELECT * FROM T
WHERE Percentage = '96%'

As we know % is a wildcard character which means zero or more ~~one~~ symbol.

While here % is not a wildcard character, it is a symbol. In such situations, in order to use '%' or '_' (underscore) as symbol and not as wildcard character we use backslash (\) before the '%' or '_'.
It will be interpreted as symbol and not as wildcard.

Example - SELECT * FROM T WHERE percentage = "96\%"

This will be interpreted as symbol and not as wildcard.

Note: backslash (\) : is called /interpreted as escape character in SQL.

2.8 Aggregate functions: COUNT, MIN, MAX, Avg, SUM

These functions compute a single value on a set of rows and returns the aggregate value.

MIN

Example :- SELECT MIN(year) FROM movies;
MIN is aggregate function which returns minimum value from column year.

⇒ It will return a single value which is the minimum year value in table movies.

Example 2: `SELECT MAX(Year) FROM movies;`

⇒ It will return a single value which is the maximum value of year in column year in table movies.

COUNT(*)

Example 3: `SELECT SUM(Year) FROM movies;`

⇒ It will return a single value which is sum of all values in column year.

Similarly, avg function will find the average of all values given in a column.

So, sum and avg are the numerical operators while min and max are comparison operators.

COUNT(*)

Example 4: `SELECT COUNT(*) FROM movies;`

⇒ Count(*) will consider all columns and count the number of rows. It will not look for distinct rows, it will just count each and every row. Hence, output will be 388269 rows.

NOTE - Count(*) consider the rows containing NULL values also.

Example 5: SELECT COUNT (year) FROM movies;

⇒ It will consider column year and will count every row without considering ~~of~~ of distinct rows.

It will output the same no. of rows as we got in Example 4 ie 388269 rows.

NOTE - Except COUNT(*), none of the aggregate function considers NULL values.

8.9 Group by :

This command is used to collect data across multiple records and group the results by one or more column.

Example 6: SELECT year, COUNT (year) FROM movies
GROUP BY year;

⇒ It will create a table with attribute year and Count all the rows having release of that year. i.e all the rows having the same year are grouped together. (grouping of all rows having some year). Here COUNT (year) will work on the group of year formed by GROUP BY clause. COUNT (year) actually gives no. of rows in that year group.

Example 7: SELECT year, COUNT (year) FROM movies

GROUP BY year ORDER BY year;

Everything will be same except result will be sorted in asc. order

Example 3: SELECT year, COUNT(year) year_count FROM movies GROUP BY year ORDER BY year_count;

→ It will generate some o/p but the rows will be ordered in ascending order of COUNT(year) or year_count (year_count is an alias). It will show the year as first row which has minimum releases (as it is ordered in ascending order) and last row of o/p will have maximum releases.

NOTE- Here we have used alias because we cannot use aggregate functions in Order by clause.

NOTE- Group_by are often used with COUNT, MIN, MAX or sum.

NOTE- If grouping column contain NULL values, all NULL values are grouped together.

8.10 Having

This command is often used in combination with the Group_by clause to restrict the groups of returned rows to only those whose the condition is TRUE.

Example 1: SELECT year, COUNT(year) year_count FROM movies GROUP BY year HAVING year_count > 1000;

→ Here, first we are grouping and from that we will get year_count and now we will print only those year groups whose year_count > 1000 i.e. only those years which have > 1000.

Here, Order of execution will be :-

Step 1: Group By year to create groups

Step 2: Now apply COUNT(year) or any aggregate function

Step 3: finally, apply the HAVING condition.

NOTE - Typically HAVING is used in combination with Group By, but it is not mandatory.

Example :- SELECT name, year FROM movies
HAVING year > 2000;

SELECT name, year FROM movies
WHERE year > 2000;

Hence, HAVING clause without Group By is same as
or works same as WHERE clause.

Example :- SELECT year, COUNT(year) year_count FROM
movies WHERE rank_score > 9 GROUP BY year HAVING
year_count > 20;
① applied over individual rows of movies table
②
③

⇒ here, in this query we have both Where and
having clause. The seq order of execution is numbered above
where is applied on individual rows while having
is applied on top of groups. If we don't have
group by, having will also be applied on individual
rows.

Having is applied after grouping while where is used
before grouping.

→ This query will return Year and COUNT(Year)
Such that, that year has more than 20 movies and
each of them have rank-score > 9.

Ph: 944-844-0102

Q.11 Order of Keywords:

While writing a sql query the order of
Keywords will be as follows:

- [1] SELECT
- [2] FROM
- [3] WHERE
- [4] GROUP BY
- [5] HAVING
- [6] ORDER BY (contains ASC and DESC after it)
- [7] LIMIT

9.1 Join and Natural Join:

Join's are used to combine data in multiple tables/relations.

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

alias of movie ↑
 alias of movie_genres ↓
 condition based on which m and g are joined

On considering fig: 31, we can see that relation movies and movies_genres are given as:-

Movies m:

Movies	
id	Name
1	n ₁
2	n ₂
3	n ₃

Genre g:

movies_genres	
movie_id	genre
1	g ₁
1	g ₂
2	g ₃
3	g ₂
1	g ₁

refer to			
id	name	year	rank-score
1	n ₁	:	:
2	n ₂	:	:
3	n ₃	:	:

A movie can have multiple genre

movie_id	genre
1	g ₁
1	g ₂
2	g ₃
3	g ₂
1	g ₁

⇒ It will print name and genre

by combining table movies m and genre g
 where m.id = g.movie_id.

Natural join:

A join where we have the same column-names across two tables.

example $T_1:$

C_1	C_2

 $T_2:$

C_1	C_3	C_4

Select * FROM T_1 JOIN T_2 ;

→ It will automatically joint T_1 and T_2 based on C_1 . It is equivalent to saying:

SELECT * FROM T_1 JOIN T_2 ON $T_1.C_1 = T_2.C_1$;

We can also write it as:

SELECT * FROM T_1 JOIN T_2 USING (C_1);

It will return C_1, C_2, C_3, C_4 in the result set.

9.2 Inner, left, right and Outer joins:

Consider the below relation set:

 $T_1:$

C_1	C_2	C_3
1	a	b
2	c	d
4	e	f
5	g	h

 $T_2:$

C_4	C_5	C_6
1	0	1
2	1	0
3	0	1
5	1	0

$\Rightarrow \text{Inner join}$

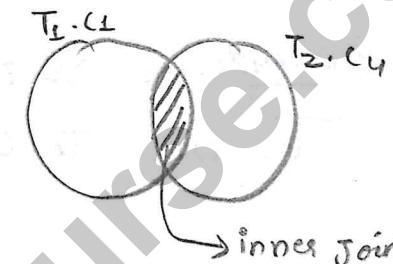
Example 1: SELECT * FROM T_1 JOIN T_2 ON $T_1.C_1 = T_2.C_4$ Ph: 844-844-0102

\Rightarrow op will be

$T \Rightarrow$

C_1	C_2	C_3	C_4	C_5	C_6
1	a	b	1	0	1
2	c	d	2	1	0
3	g	h	5	1	0

Inner join diagrammatically :-



\Rightarrow Left + outer join:

Example 2: SELECT * FROM T_1 LEFT OUTER JOIN T_2
ON $T_1.C_1 = T_2.C_4$

\Rightarrow op will be

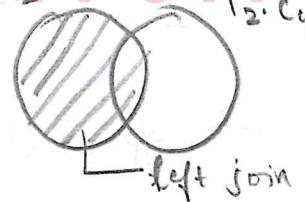
$T \Rightarrow$

No match
for this
in right relation
hence, we have
added NULL for C_4, C_5, C_6

C_1	C_2	C_3	C_4	C_5	C_6
1	a	b	1	0	1
2	c	d	2	1	0
4	e	f	NULL	NULL	NULL
5	g	h	5	1	0

here, we are combining the left relation with right relation if there are matching rows and if no matching rows are found the left relation row is clubbed with NULL

Left outer join diagrammatically \Rightarrow



NOTE - We can either use keyword LEFT OUTER JOIN or LEFT JOIN, both are same.

Right Outer join:

Example 3: SELECT * FROM T₁ RIGHT OUTER JOIN T₂ ON T₁.C₁ = T₂.C₄

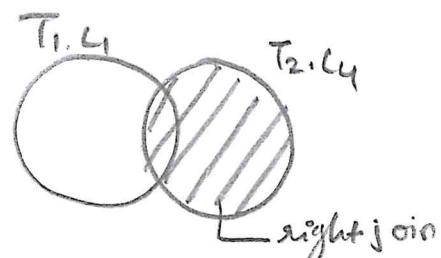
\Rightarrow o/p will be:

T ₁	C ₁	C ₂	C ₃	C ₄	C ₅	C ₆
1	a	b	f	g	0	1
2	c	d	2	2	2	0
5	g	h	5	1	0	0
Null	Null	Null	3	0	1	0

In This row eight relation do not have

$T_1.C_1 = T_2.C_4$ hence, we have filled C₁, C₂, C₃ with NULL

Right outer join diagrammatically \Rightarrow



NOTE - We can either use keyword RIGHT OUTER JOIN or RIGHT JOIN, both are same.

$\rightarrow \leftarrow$ Full outer join:

Example 4: `SELECT * from T1 FULL OUTER JOIN T2 ON T1.C1 = T2.C4`

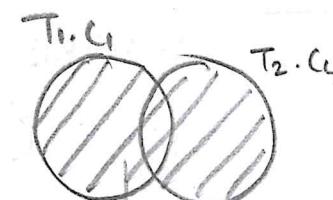
\Rightarrow o/p will be:

Left join \cup

Right join

	C ₁	C ₂	C ₃	C ₄	C ₅	C ₆
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

Full outer join diagrammatically \Rightarrow



NOTE- We can use either FULL OUTER JOIN or FULL JOIN while writing a query, both are same.

NOTE- INNER JOIN \equiv JOIN (Both are same)
(inner is optional keyword here)

NOTE- We can have 3-way join (Joining 3 relations) and K-way joins (Joining K relations)

Example- `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 WHERE a.id=1`

This query is combining 3 relations in a SQL query.

NOTE - Joins are the most expensive computationally when we have large tables. It is one of the famous area of research that how to reduce its computation time.

SUB QUERIES / NESTED QUERIES / INNER QUERIES

Syntax:

SELECT <column-names> FROM <table-names>
WHERE <column-name> OPERATOR
(SELECT <column-name> FROM
<table-name> WHERE);

OR

SELECT Column-name [, Column-name]
FROM Table1 [, Table2]
WHERE Column-name OPERATOR (SELECT
Column-name [, Column-name]
From Table1 [, Table2] [WHERE]);

IN

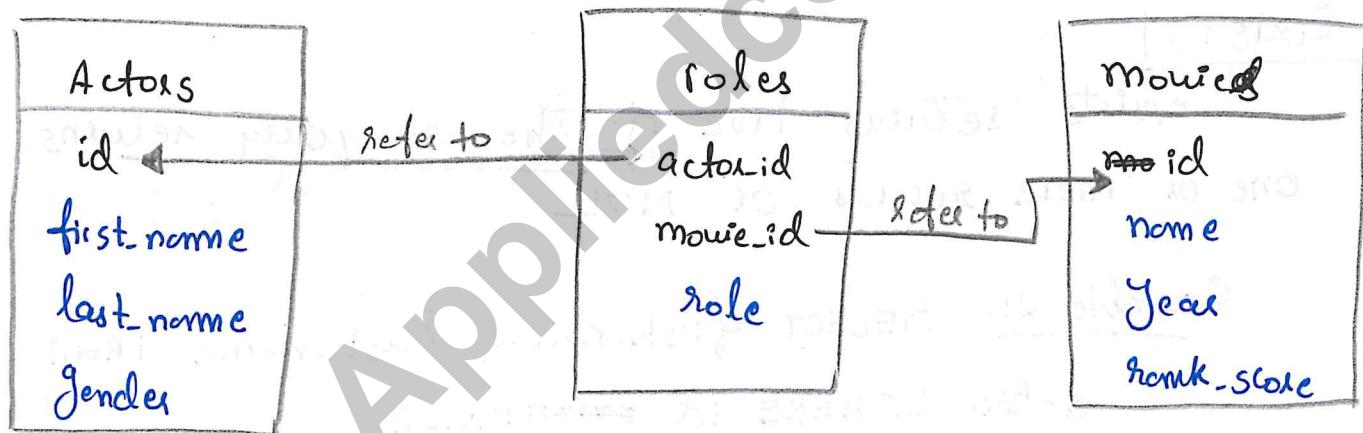
Example: list all actors in the movies Schindler's list

⇒ Note that schindler's list is the whole series of movies.

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" } } } nested
 Keyword to give a select value from a set of list

→ here we are accessing 3 relations in a query using sub-queries. One way to do this is by writing it in the form of subquery while another way to solve this is using "Join's". It will be discussed later.

This could be understood as:



It will first execute the innermost query (Q_3) ie "SELECT id FROM movies WHERE name = "Schindler's List";"

⇒ It will result in the set of 126 tuples.

Next query which will get executed is Q_2 ie "SELECT actor-id FROM roles WHERE movie-id IN (126 tuples);"

⇒ It will return set of actor-id's.

At last, the outermost query (Q_1) got executed
i.e "SELECT first-name, last-name FROM actors
WHERE id IN (set of all actor id's);

⇒ It will op all first-name, last-names whose
~~the set~~ actor id is in set op of Q_2 :

NOTE

In SQL 'IN' can be thought as belongs to (\in)
in set theory. While 'NOT IN' can be
thought as do not belongs to (\notin) in set
theory.

EXISTS

Exists returns true if the subquery returns
one or more records OR NULL

Example 2: SELECT first-name, last-name FROM
actors WHERE id ~~exist~~ EXISTS

(SELECT actor-id FROM Roles WHERE movie-id
EXISTS (SELECT id FROM movies WHERE
name = "Schindler's list")));

→ If the subqueries return a non-empty
set say $\{f_1\}, \{f_2\}$, having one or more than
1 element OR return a NULL than ~~what~~ we
will get whatever we want in the result set
of outer query.

NOTE Similarly it works for NOT EXISTS.

All operators returns TRUE if all of the subquery values meet the condition.

Example 3: SELECT * FROM movies WHERE

rank-score >= ALL (SELECT MAX(rank-score)
FROM movies);

will return true if all subquery meet condition.

⇒ It will get us all movies whose rank-score is same as the maximum rank-score.

Here, subquery will return only one value which is max(rank-score) = {9.9}. Now outer query will print/select all columns ~~to~~ from table movies where rank-score >= {9.9}.

NOTE- Subqueries / inner queries are more easy to read and write when compared to Join's.

ANY

Any operator returns TRUE if any of the subquery values meets the condition.

* Correlated sub queries \Rightarrow

In SQL, Correlated subquery / synchronized 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 efficient.

Example:

```
SELECT emp-no, name FROM Employees emp
WHERE salary > (SELECT AVG(salary)
                  FROM Employees
                  WHERE department =
                        emp.department);
```

\Rightarrow Here, inner query will return :- it is going to compute average salary of employees and finally will print those emp-no and names of employees whose salary is greater than average salary of employees within that department. (we know this fact of within that department because we are using alias emp in our inner query).

i.e. per department list all the employees whose salary's are greater than avg salary in that department.

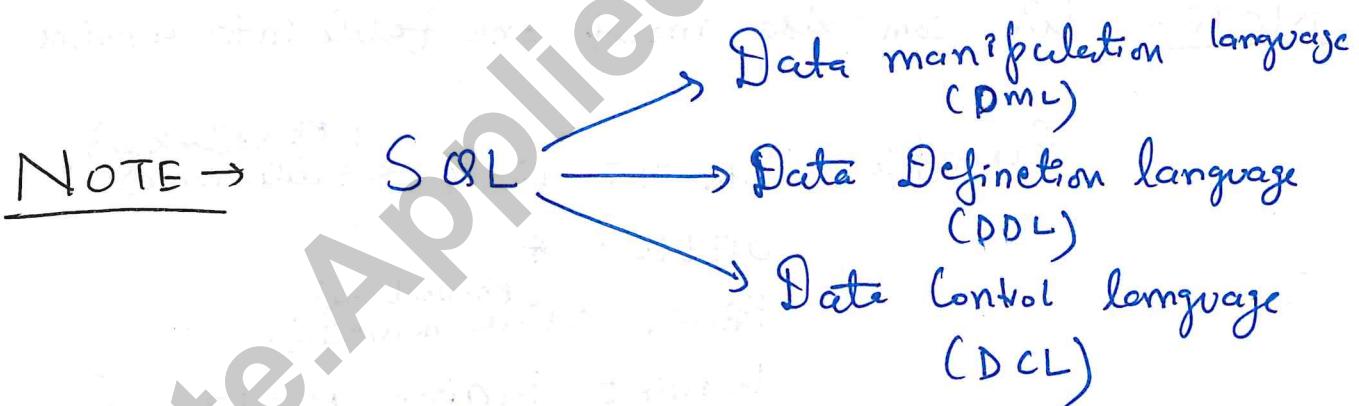
QnB
It is called Correlated Subquery because for every employee in outer query we need to run inner subquery again and again.

So, here we using info of outer query in the inner query and this inner query needs to be run for every employee.

NOTE- Correlated queries are computationally expensive.

DATA MANIPULATION LANGUAGE

1.1 Insert:



DML includes SELECT, INSERT, UPDATE, DELETE.

These four commands ~~are~~ belongs to data manipulation language.

INSERT is used to insert data / row / tuples in a relation.

Syntax- `INSERT INTO <table-name> (<attribute-names>
VALUES (<values-as-per-attribute-sequence>);`

Example:

INSERT INTO movies (id, name, year, rank-score)
VALUES

inserting multiple rows in relation Movies

{ (412321 , 'THOR' , 2011, 7), → 1st row
(412323, 'IRON MAN 2' , 2010, 7); → 2nd row }

NOTE - Since 'id' is the primary key in table "movies" So, we cannot insert two rows with same 'id' value.

NOTE - We can also insert one table into another

Example INSERT INTO <table-name> (Phone-book_2)
SELECT *
FROM <table-name>
WHERE NAME IN ('John Doe',
 'Peter Doe')

⇒ It will copy all rows from Phone-book to phone-book_2 whose names are either ' John Doe ' or ' Peter Doe '.

11.2 Update, Delete:→ ↪ Update:

Syntax: UPDATE <Table-name> SET
Col1 = val1, Col2 = val2 WHERE
<Condition>;

Example -

UPDATE movies SET rank-score = 9
WHERE id = 412321;

→ It will update the rank-score value of
the tuple whose id = 412321.

We can also update multiple column values by
separating them using commas or using subquery.

→ ↪ Delete:

Syntax: DELETE FROM <table-name>
WHERE <Condition>;

Example:-

DELETE FROM movies WHERE
id = 412321;

→ It will delete the row which is having
id = 412321

NOTE - If we want to delete all rows from table
we use a command called TRUNCATE. It is not
DML it is a DDL. Its Syntax is: TRUNCATE TABLE
<tablename>;

NOTE- We can also delete all rows from a table using 'DELETE' command. This could happen if we remove WHERE clause.

Syntax - DELETE FROM <table-name>;

DATA DEFINITION LANGUAGE (DDL)

12.1 Create Table:

- * DDL helps us to define, alter, drop a table.
- * To create a new table in database is created as follows:

Syntax: CREATE TABLE <table-name>(
 <attribute-name> <datatype> <Specification>);

Example

CREATE TABLE language (

 id INT PRIMARY

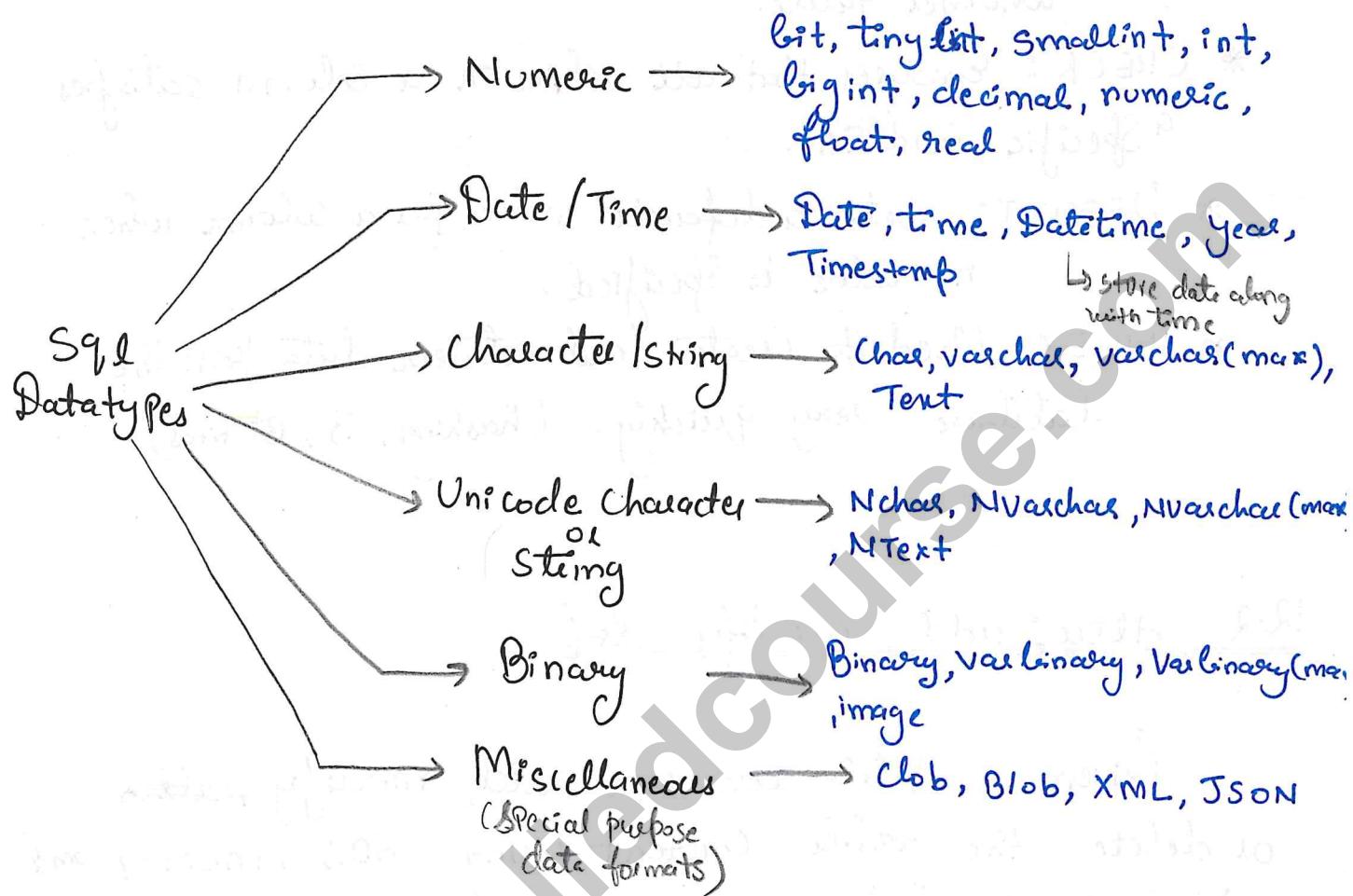
 → it is integer type

 lang VARCHAR(50) NOT NULL);

 → it is a character type

⇒ If we try to insert any duplicate id or lang as NULL, it will throw an error.

→ SQL Data Types:



Note that, not every database support all data types, Some may support some data types while other may support other data types.

→ Constraints / Properties:

* NOT NULL: Ensures that a column cannot have a NULL value.

* UNIQUE: Ensures that all values in a column are different.

* PRIMARY KEY: A combination of NOT NULL and UNIQUE.

Mail: gatecse@appliedcourse.com

- * FOREIGN KEY: Uniquely identifies a row/record in another table.
- * CHECK: Ensures that all values in a column satisfies a specific condition.
- * DEFAULT: Sets a default value for a column when no value is specified.
- * INDEX: Used to create and retrieve data from the database very quickly. (hashing, B, B+ trees)

12.2 Alter; add , modify, drop :

Given a table we can add, modify ~~and delete~~ or delete the table using ADD, MODIFY and DROP commands in DDL

→ ↴ ↵ ADD : It helps us to add new columns or constraints in DB

Syntax: ALTER TABLE <table_name>
 ADD <column_name> <data-type>
 <constraints>;

Example ALTER TABLE language ADD
 Country VARCHAR(50);

→ ↴ ↵ MODIFY: It helps to modify column name, datatype or constraints.

Syntax: ALTER TABLE <table_name> MODIFY

 <column_name> <data-type> <constraints>;

Example-

ALTER TABLE language MODIFY country
VARCHAR (60);

→ Drop: It will ^(delete) drop the mentioned column.

Example ALTER TABLE language DROP country;

→ It will remove whole column "country".

12.3 Drop Table, Truncate, Delete:

→ Drop table: It will remove both the table and all of the data permanently.

Syntax: DROP TABLE <table name>;

Example: DROP TABLE language;

It will wipe off whole table, while delete command is use to delete some or all rows of table.
It will never delete the table structure.

NOTE- We can also add one modification to it

DROP TABLE <table name> IF EXISTS;

→ It will drop the table (whole table) if it exists in database. So that we don't get any error if we don't know whether it exist or not.

Truncate: It will remove the content of table but not table structure.

Syntax - TRUNCATE TABLE <table-name>;



DELETE FROM <table-name>

DATA CONTROL LANGUAGE (DCL)

13.1 Grant, Revoke:

In Database, different people have different access to different parts of data because same database is being accessed by multiple people in an organization. Hence, a developer, data scientist, admins etc all have different access permission of the database.

may have permission to add, delete, modify etc

can only access the data (read). (cannot modify it)

have permission to create, insert relation in a database.

DCL helps us to control who can access what. It is not implemented by all databases.

GRANT :

It allocates specified users to perform specified tasks.

Example :- GRANT ALL ON db1.* To "jeffrey @'localhost';

↓
all
stands
for all
permissions

db1 is
the name
of database
and * stands
for all tables
of database.

Example 2 :- GRANT 'role 1', 'role 2' To 'user1@'localhost',
'user2 @'localhost';

here these are different roles
like admin etc which is
given to user 1 and user 2.

Example 3: GRANT SELECT ON world.* To 'role3';

it has given
Select Permission
on database named
world over all
of its tables / relations

REVOKE :

It cancels previously granted or denied permissions.

Example 1: REVOKE INSERT ON *.* FROM
"jeffrey@'localhost';

all database and
all tables of each database

Example 2: REVOKE SELECT ON world.* FROM 'role 3';

⇒ Revoking Select permission from role 3 on database named world over all of its tables.

NOTE- Best place to learn sql is "reference manual" of any database we are using.

SQL : EXAMPLES AND SOLVED PROBLEMS

(ii) A relational database contains two tables student and performance as shown below:

Student:

Role no	StudentName
1	Amit
2	Priya
3	Rohan
4	Smita
5	Vinit

Performance:

Roll_no	Subject Code	Marks
1	A	86
1	B	95
1	C	90
2	A	89
2	C	92
3	C	80

The primary key of the student table is Roll_no. For the performance table, the columns Roll_no and Subject_code together form the primary key. Consider the SQL query given below:

```
SELECT S.student_name, sum(P.marks)
From Student S, Performance P
WHERE P.marks > 84
GROUP BY S.student_name;
```

The number of rows returned by the above SQL query is _____

⇒ Given the query:

```
SELECT S.student_name, sum(P.marks)
```

executed 1st {From Student S, Performance P} → This indicates cross product of S and P
executed 2nd {WHERE P.marks > 84} except 6th row in "Performance" is satisfied.
executed 3rd {GROUP BY S.student_name;} → this will generate 5 groups with 5 distinct student name

∴ group by will create 5 groups for 5 distinct student name hence, no. of rows/tuples = 4

(Q2) Consider the following two tables and four queries in SQL.

Book (isbn, bname), Stock (isbn, copies)

Query 1:

```
SELECT B.isbn, S.copies  
FROM Book B INNER JOIN Stocks  
ON B.isbn = S.isbn;
```

Query 2:

```
SELECT B.isbn, S.copies  
FROM Book B LEFT OUTER JOIN Stocks  
ON B.isbn = S.isbn;
```

Query 3:

```
SELECT B.isbn, S.copies  
FROM Book B RIGHT OUTER JOIN Stocks  
ON B.isbn = S.isbn;
```

Query 4:

```
SELECT B.isbn, S.copies  
FROM Book B FULL OUTER JOIN Stocks  
ON B.isbn = S.isbn;
```

Which one of the queries above is certain to have an output that is superset of the o/p's of the other three queries?

- (A) Query 1
- (B) Query 2
- (C) Query 3
- (D) Query 4

As we know tuples we get in full outer join is union of $B \bowtie S \cup B \bowtie S \cup B \bowtie S$

Hence, Collect optim is the query with FULL OUTER JOIN ie Query 4 (D).

(Q43) Consider a database that has the relational Schema Emp (EmpID, EmpName, DeptName). An instance of the schema Emp and a SQL query on it are given below:

Emp:

EmpID	EmpName	DeptName
1	X4A	AA
2	X4B	AA
3	X4C	AA
4	X4D	AA
5	X4E	AB
6	X4F	AB
7	X4G	AB
8	X4H	AC
9	X4I	AC
10	X4J	AC
11	X4K	AD
12	X4L	AD
13	X4M	AE

Query —

```

SELECT Avg(EC.Num)
FROM EC
WHERE (DeptName, Num) IN
    (SELECT DeptName,
    COUNT(EmpID) AS
    EC(DeptName, Num)
    FROM Emp
    GROUP BY DeptName);
    
```

The output of executing the SQL query is —

⇒ Here the Nested query is:

SELECT AVG(Ec.Num)

FROM EC

WHERE (DeptName) IN (

SELECT DeptName,
COUNT(EmpID) AS
Ec(DeptName, Num)
FROM Emp
GROUP BY DeptName);

This will
be
executed
first

This is the alias
for the new
table created
by the inner query

This table EC has
DeptName and Num

→ Count(EmpID)

The inner query will be executed first. It
will group by the based on DeptName i.e.
AA, AB, AC, AD, AE and Count(EmpID) will
give no. of employees in each department. Hence output
of inner query will be:

Ec:

DeptName	Num
AA	4
AB	3
AC	3
AD	2
AE	1

Now the outer query is asking for Avg on
Num from resultant relation EC.

ie $\frac{4+3+3+2+1}{5} = \frac{13}{5} = 2.6$.

- (Q4) Consider the following database table named
top_scores:

Top_scores:

Player	Country	goals
Klose	Germany	16
Ronaldo	Brazil	15
G Muller	Germany	14
Fontaine	France	13
Pele	Brazil	12
Klinsmann	Germany	11
Kocsis	Hungary	11
Batistuta	Argentina	10
Cubillas	Peru	10
Lato	Poland	10
Hinckes	England	10
T Muller	Germany	10
Rahn	Germany	10

Consider the following query:

① { SELECT ta. Player FROM top-scores AS ta
WHERE ta.goals > ALL (SELECT tb.goals
FROM top-scores AS tb
WHERE tb.country = 'Spain')
AND ta.goals > ANY (SELECT tc.goals
FROM top-scores AS tc
WHERE tc.country = 'Germany'));

The number of tuples returned by the above SQL query is _____

⇒ Here we have 2 sub queries/nested query's.

~~First we will~~ Before start solving the innermost query (③) if we look at the first sub query (②) we can see that it will result empty set because there is no country = "Spain" in our table given. Therefore, it will return an empty set.

Hence, it will always be true because $\text{ALL}(\text{empty set})$

||
TRUE

Since, there is logical AND between ② and ③
∴ ② is already TRUE, so now we need to execute ③

The inner most query (③) will return:

The goals made by players whose country = "Germany".

O/P →

te. goals
16
14
11
10
10

The condition before logical AND is
 $tq.\text{goals} > \text{Any } \{ 16, 14, 11, 10 \}$ ie $tq.\text{goals}$
 Should be greater than 10.

Since query ② is ignored as it is TRUE ∴ outer
 query will print Player names whose goals are > 10
 Hence, no. of rows returned $\underline{\underline{=7}}$

(Q5) Consider the following database table named
water_schemes:

water_schemes:

Scheme_no	district_name	Capacity
1	Ajmer	20
1	Bikaner	20
2	Bikaner	10
3	Bikaner	20
1	Churu	10
2	Churu	20
1	Dungarpur	10

The number of tuples returned by the following SQL query is:

With total (name, capacity) as
 ① { Select distinct_name , sum (capacity)
 From water_schemes
 group by district_name
 With total_avg (capacity) as
 { Select avg(capacity) @
 from total
 } }
 ② { Select name
 from total, total_avg
 Where total.capacity >= total_avg.capacity
 } }
 ③ { }

- (A) 1
- (B) 2
- (C) 3
- (D) 4

→ With is a keyword introduced by oracle. Never mysql version also uses "with".

With creates a temporary relation / table with the result returned by a query and using the given table and attributes name.

Here, Q1 will return \Rightarrow Total:

This is a temporary table which is accessed by the query coming part of further.

name	Capacity
Ajmer	20
Bikaner	40
Churu	30

Sum of Capacity of each city

This name is given by "With"

APPLIED COURSE NOTE - Using "with" we can access the resultant table as it get saved in memory **Ph: 844-844-0102** 151

The output of φ_2 will be: (This query is using table)

Total-aug: Created by φ_2)

New table
Created by
 φ_2

Capacity
25

$$\begin{aligned} \text{avg of } & \\ \frac{20+40+30+10}{4} & \\ = 25 & \end{aligned}$$

The output of φ_3 will be: It will use both total and total-aug to get the result. It is performing Cartesian product of total and total-aug and returns the tuples whose capacity is greater than average capacity 25.

hence, only Likanee and Chuu are having capacity 30 and 40 respectively which is greater than 25
Therefore, no. of tuples returned = 2 //

15.2 Solved problems - 2 :

(Q6) Select operation in SQL is equivalent to

- (A) The selection operation in relational algebra
- (B) The selection operation in relational algebra, except that select in SQL retains duplicates
- (C) The projection operation in relational algebra
- (D) The projection operation in relational algebra, except that select in SQL retains duplicates.

As we know select operation in SQL selects columns while selection operation in RA selects rows/tuples. While projection in RA selects columns but it removes duplicates but select set in SQL retains duplicates.
 Hence, correct option is (D).

(Q 7) Consider the following relations:

Students:

<u>Roll_no</u>	<u>Student_Name</u>
1	Raj
2	Rohit
3	Raj

Performance:

<u>Roll_no</u>	<u>Course</u>	<u>Marks</u>
1	Math	80
1	English	70
2	Math	75
3	English	80
2	Physics	65
3	Math	80

SELECT S.Student_Name, Sum(P.marks)

FROM Student S, Performance P }

WHERE S.Roll_No = P.Roll_No }

GROUP BY S.Student_Name;

The number of rows that will be returned by the SQL query is _____

This is equivalent
to JOIN

- (A) 0
- (B) 1
- (C) 2
- (D) 3

⇒ Since, here we are using group by using Student_name and we

(Q8) Consider the following relation:

Cinema (theater, address, capacity)

Which of the following options will be needed at the end of the SQL query:

SELECT P₁.address

FROM Cinema P₁

Such that it always find the addresses of theaters with maximum capacity?

- (A) WHERE P₁.Capacity >= ALL (SELECT P₂.Capacity FROM Cinema P₂)
- (B) WHERE P₁.Capacity >= ANY (SELECT P₂.Capacity FROM Cinema P₂)
- (C) WHERE P₁.Capacity > ALL (SELECT MAX(P₂.Capacity) FROM Cinema P₂)
- (D) WHERE P₁.Capacity > ANY (SELECT MAX(P₂.Capacity) FROM Cinema P₂)

⇒ 'Any' will not work here because even if there is even 1 theater whose capacity is max it will get picked up. So, only options left are (A) and (C).

When I'm selecting maximum capacity of cinema hall nothing could be greater than it so (C) will not work because ALL will return only one value (which is maximum value) and ∴ it should be \geq and not $>$

Hence, correct and feasible option is (A)

(Q9) Given the following statements:

S₁: A foreign key declaration can always be replaced by an equivalent check assertion in SQL.

S₂: Given the table R(a, b, c) where a and b together form the primary key, the following is a valid table definition.

CREATE TABLE S (

a Integer,

d Integer,

e Integer,

PRIMARY KEY (d),

FOREIGN KEY (a) REFERENCES R)

Which one of the following statement is CORRECT?

- (A) S₁ is TRUE and S₂ is FALSE
- (B) Both S₁ and S₂ are TRUE
- (C) S₁ is FALSE and S₂ is TRUE
- (D) Both S₁ and S₂ are FALSE

→ S₁: It is not always delete because we do have Cascade option on delete or update.

S₂: In table S, primary key is d and foreign key a, should point to primary key of R. In R (a,b) is a primary key not 'a' alone is a primary key. So, 'a' in table S should refer to (a,b) and not only to 'a'. Hence, it is false

Mail: gatetce@appliedcourse.com Both are false.

(Q10) Given the following Schema:

employees (emp_id, first_name, last_name, hire_date, dept_id, salary)

department (dept_id, dept_name, manager_id, location_id)

You want to display the last names and hire dates of all latest hires in their respective departments in the location ID 1700. You issue the following query:

SQL > SELECT last_name, hire_date

FROM employees
WHERE (dept_id, hire_date) IN
(SELECT dept_id, MAX(hire_date)
FROM employees JOIN departments USING(dept_id)
WHERE location_id = 1700
GROUP BY dept_id);

inner query { }

What is the outcome?

- (A) It executes but does not give the correct result.
- (B) It executes and gives the correct result.
- (C) It generates an error because of pairwise comparison.
- (D) It generates an error because of the GROUP BY clause cannot be used with table joins in a subquery.

⇒ Since there is no logical and syntactical error in query therefore, it will execute and o/p generated by the query is same as what given in english statement. Hence, it will execute and will give correct answer. Therefore, (B) is the correct answer.

15.3 Solved Problem - 3 :

(Q 11) SQL allows tuples in relations, and correspondingly defines the multiplicity of tuples in the result of joins. Which of the following queries always gives the same answer as the nested query shown below:

SELECT * FROM R WHERE A IN (SELECT S.a FROM S)

- (A) SELECT R.* FROM R, S WHERE R.a = S.a
- (B) SELECT DISTINCT R.* FROM R, S WHERE R.a = S.a
- (C) SELECT R.* FROM R, (SELECT DISTINCT A FROM S) AS S1 WHERE R.a = S1.a
- (D) SELECT R.* FROM R, S WHERE R.a = S.a AND IS UNIQUE R.



Let us consider a sample relation instance of R and S:

R:

a
1
1
2
2
3

S:

a
1
1
2
2

So, the o/p of original query will be:

a
1
1
2
2

Option A will return:

a
1
1
1
1
2

Ph: 844-844-0102

Option B will return:

a
1
2

Option C will return:

a
1
1
2
2

It will do join

a
1
2

a
1
2
3

Hence, correct option is (C)

- (Q 12) Consider the following relational schema:
Employee (empid, empName, empDept)
Customer (custid, custName, salesRepId, rating)
SalesRepId is a foreign key referring to empid of the employee relation. Assume that each employee makes a sale to at least one customer. What does the following query return?

SELECT empName

From Employee E

WHERE NOT EXISTS (SELECT custid

From Customer C

WHERE C.SalesRepId = E.empid
AND C.Rating <> 'Good');

- (A) Names of all employees with at least one of their customers having a 'Good' rating.
- (B) Names of all employees with at most one of their customers having a 'Good' rating.
- (C) Names of all the employees with none of their customers having a 'Good' rating.
- (D) Names of all the employees with all their customers having a 'Good' rating.

⇒ The assumption made in question "Each employee makes a sale to at least one customer" means that in SalesRepId, every empId exists at least once.

The inner query will give : CustId for SalesRepId = empId and Rating is not equal to "good".
Outside the inner query, we have NOT EXISTS . So, its kind of like negation over negation (as in inner query we have Rating != "Good"). So, any employee who have even 1 customer whose Rating is not equal to good will not be there in the result set.
So, option A, B and C are wrong. Hence, D is the most appropriate option.

(Q 13) Consider the following tables A, B and C:

Table A :

Id	Name	Age
12	Arun	60
15	Shreya	24
99	Rohit	11

Table B :

Id	Name	Age
15	Shreya	24
25	Hari	40
98	Rohit	20
99	Rohit	11

Table C :

Id	Phone	Area
10	2200	02
99	2100	01

How many tuples does the result of the following SQL query contains?

```
SELECT A.Id  
FROM A  
WHERE A.age > ALL (SELECT B.age  
                      FROM B  
                     WHERE B.name = 'arun')
```

- (A) 4
- (B) 3
- (C) 0
- (D) 1

⇒ The inner sub query will return empty set and we know $\text{ALL}(\text{empty set}) = \text{True}$.
 Hence, $A.\text{age} > \text{ALL}()$ is true for all tuples.
 and it will return $A.\text{id}$ for all $A.\text{age} > \text{all}$. Therefore, it will return 3 tuples.

(Q14) Database table by name loan-Records is given below:

Borrower	Bank_manager	loan_Amount
Ramesh	Sunderajan	10000.00
Suresh	Ramgopal	5000.00
Mahesh	Sunderajan	7000.00

What is the o/p of the following SQL query?

```
SELECT COUNT(*)
FROM ((SELECT Borrower, Bank_Manager
       FROM Loan_Records) AS S
      NATURAL JOIN (SELECT Bank_manager, loan_Amount
                    FROM Loan_Records) AS T);
```

- (A) 3
- (B) 9
- (C) 5
- (D) 6

⇒ According to the query we will have

S:

Borrower	Bank_manager
Ramesh	Sunderajan
Suresh	Ramgopal
Mahesh	Sunderajan

NATURAL JOIN

T:

Bank_manager	Loan_Amount
Sunderajan	10000.00
Ramgopal	5000.00
Sunderajan	7000.00

It will result in distinct 5 tuples :

Borrower	Bank_manager	loan_Amount
Ramesh	Sunderajan	10k
Ramesh	Sunderajan	7k
Buvesh	Rangopal	5k
Mahesh	Sunderajan	10k
Mahesh	Sunderajan	7k

5
distinct
tuples

Hence, output will be 5 tuples (as it's Count(*))