0th

SQL COMMANDS

8.1 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~~ 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 ⇒

Field	Type	Null	Key	default	Extra
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	

Data type

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 shaurkh have last name 'Khan'

8.2 Select :

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

Given a query " SELECT * FROM movies; "

Keywords

- SELECT → it says select all columns metioned.
- * → list of columns. Here * means everything
- FROM → from is a keyword which metion the table_name from which data is to
- movies → table_name

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- Describe movies will only result in the description of every attribute present in table movies. While, select basically ~~sabi~~ 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 388,209 row as o/p ~~cnt~~ with only two attributes "name and years". 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 ~~exp1~~ 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).

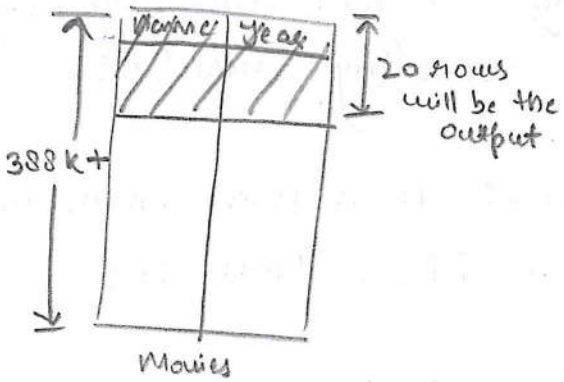
So, it is suggested to query only those attributes which are required rather than using *.

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 1: SELECT name, year FROM movies LIMIT 20;
↳ # 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 an 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."

Example 2: SELECT name, year FROM movies LIMIT 20, OFFSET 20;

o/p ⟹ 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.

8.4 Order by:

This keyword is used to get the ordered output for a given ~~column~~ 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;

- year → The result is ordered based on year
- DESC → 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.

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:

Distinct will work/applied on both of these.

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

It will give 86844 rows as o/p, there are ~~is~~ two row with first_name = "Zvonko" because there are two "Zvonko" with different last name ie '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:

WHERE:

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

Example 1: SELECT name, year, rankscore FROM movies WHERE rank_score > 9;

WHERE → 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 their name, year and rank_score.
It will output 1069 rows which are rated > 9.

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

→ It will give top 20 movies (name, year, rank_score), where rank_score > 9.

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 =, <>/!= (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 / doesnot exist

NULL is a special keyword in SQL. for any attribute value as NULL means that value doesnot exists.

NOTE The operator " =" doesnot work with NULL

Example 1: SELECT name, year, rankscore ~~to~~ 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, rankscore FROM movies WHERE rank_score IS NULL; LIMIT 20;

↓

it will work to give all rows where rank_score is NULL

- It will o/p 20 rows with rank_score as NULL

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

Will o/p all rows where rankscore 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, rankscore FROM movies WHERE rankscore > 9 OR year > 2007;

condition 1 — condition 2

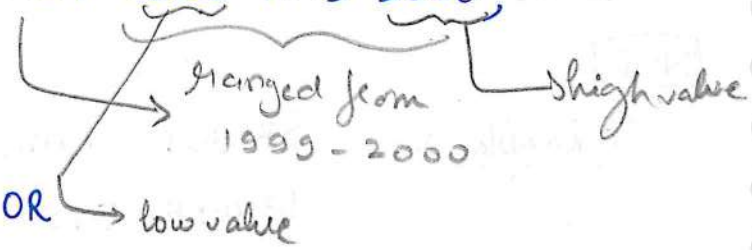
Combined condition 1 and 2 and will result to TRUE if any of the condition gets true.

⇒ It will list all movies that are either released after 2007 ~~and~~ or having rank_score > 9.

NOTE- ALL and ANY are discussed in topic Subqueries.

BETWEEN

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



(without between) SELECT name, year, rankscore 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 ie 1999 and 2000 are included in result.

Example 5: SELECT name, year, rankscore 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 <= high value otherwise we will get an empty set as result.

IN

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~~ characters)

⇒ 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~~ actors WHERE firstname 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';

('_' (underscore) 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 = '9%'

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 wild card character we use backslash (\) before the '%' or '_'

Example - SELECT * FROM T WHERE percentage = "9[6]\%"

This will be interpreted as symbol and not as wild card.

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

8.8 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 1: SELECT MIN(year) FROM movies;

MIN(year): MIN = aggregate function; year = column name on which aggregate function is applied

⇒ 97 will return a single value which is the minimum year value in table movies.

MAX

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.

SUM

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 an 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.

COUNT

Example 5: SELECT COUNT (year) FROM movies;

⇒ It will consider column year and will count every row without considering ~~do~~ 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.

P.9 Group by:

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

Example 1: 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. ie all the rows having the same year are grouped together. (Grouping of all rows having same 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 2: SELECT year, COUNT(year) FROM movies
GROUP BY year ORDER BY year;

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

It is an alias/another name for count(year)

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

⇒ It will generate same 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 ie 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 aggregate function

Step 3: finally, apply the HAVING condition.

NOTE- Typically HAVING is used in combination with GROUP BY, but it is not mandatory.

~~Eg:~~ 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 2: SELECT year, COUNT(year) year_count FROM movies WHERE rank_score > 9 GROUP BY year HAVING year_count > 20;

(3) (1) (2) (4)

(1) → applied over individual rows of movies table

⇒ here, in this query we have both Where and having clause. The seq order of execution is numbered above

NOTE 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.

Q.11 Order of keywords:

While writting 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

JOIN'S

9.1 Join and Natural Join:

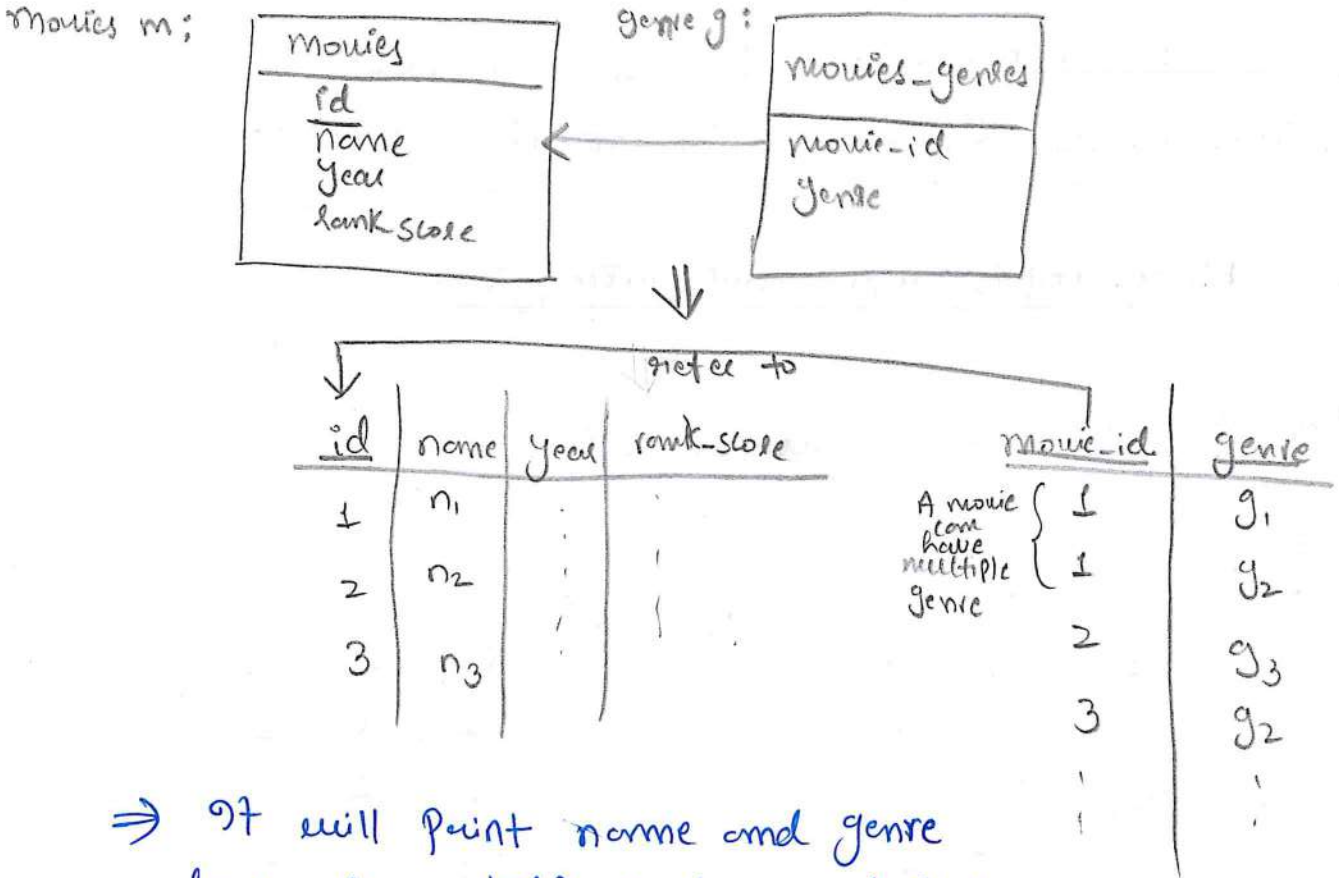
Join's are used to combine data in multiple tables/relations.

Example 1:

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

(alias of movies; 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:-



⇒ 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 accross 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 (C1);

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

Fig: 32

Inner join:

Example 1: SELECT * FROM T_1 JOIN T_2 ON $T_1.C_1 = T_2.C_4$

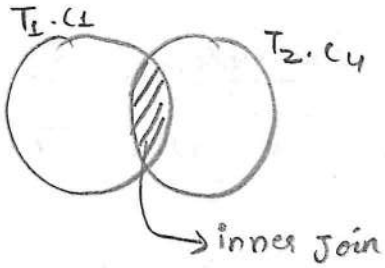
⇒ o/p will be

T ⇒

$C_1 = C_4$

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

Inner join diagramatically :-



left outer join:

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

⇒ o/p will be

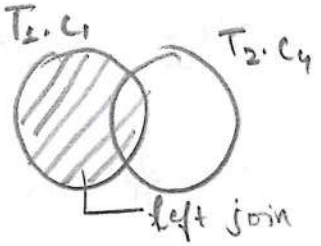
T ⇒

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

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

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 diagramatically ⇒



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

* Right Outer join:

Example 3: SELECT * FROM T_1 RIGHT OUTER JOIN T_2 ON $T_1.C_1 = T_2.C_4$

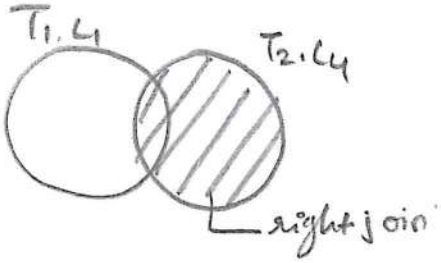
⟹ o/p will be:

T=

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

← In this row right relation do not have $T_1.C_1 = T_2.C_4$ hence, we have filled C_1, C_2, C_3 with NULL

Right outer join diagramatically ⇒



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

Full outer join:

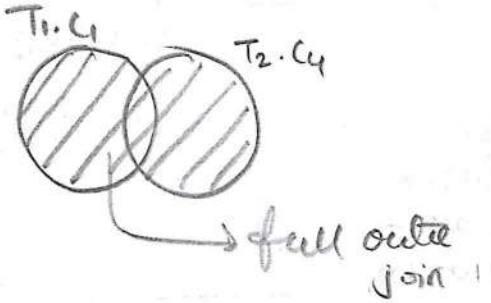
Example 4: SELECT * FROM T_1 FULL OUTER JOIN T_2 ON $T_1.C_1 = T_2.C_4$

$\Longrightarrow$ 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

This query is combining 3 relations ~~at~~ 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 ~~time~~ 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);

NOTE: OPERATORS here are:- IN, NOT IN, EXISTS, NOT EXISTS, ANY, ALL etc.

OR

```
SELECT  Column_name [, Column_name]
FROM  Table1 [, Table2]
WHERE  Column_name OPERATOR ( SELECT
        Column_name [, Column_name]
       FROM  Table1 [, Table2] [WHERE]);
```

IN

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

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

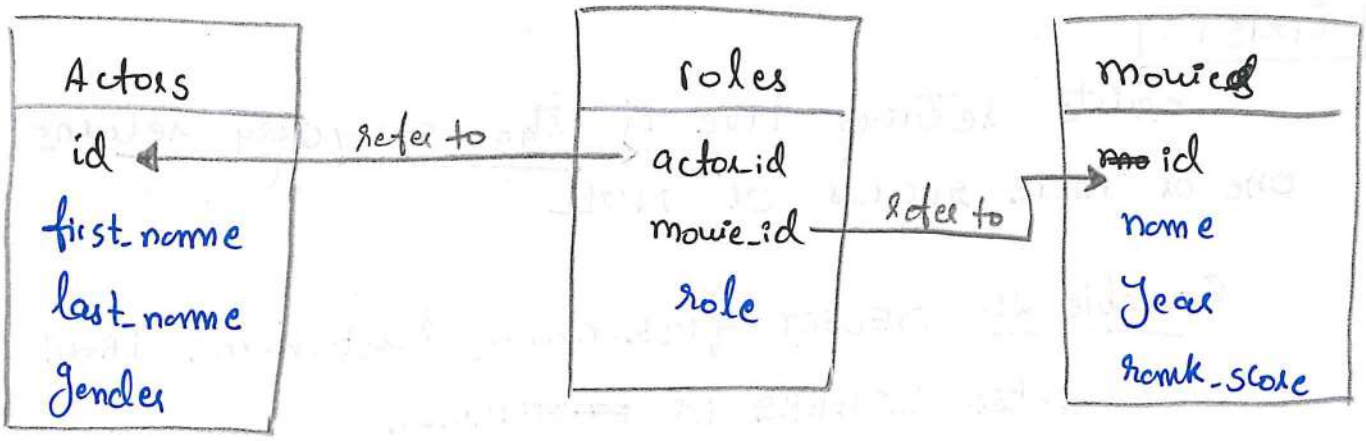
Q_1

SELECT first_name, last_name FROM actors WHERE id IN (Q_2 SELECT actor_id FROM roles WHERE movie_id IN (Q_3 SELECT id FROM movies WHERE name = "Schindler's List"));

nesting

Keyword to give a select value from a set or list

⟹ Here we are accessing 3 relations in a query using sub-queries. One way to do this is by writting it in the form of sub query while another way to solve this is using "Join's". ~~It will be used as:~~ 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 outer most query (Q_1) got executed ie " SELECT first_name, last_name FROM actors WHERE id IN ({ set of all actor id's});

⇒ It will o/p all first_name, last_names whose ~~the set~~ actor id is in set o/p 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 sub queries return a non-empty set ~~say {1}, {1}~~, 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

All operator 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 ~~of~~ 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 ~~retu~~ values meets the condition.

* Correlated sub queries ⇒

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 );
```

(inner query)

⇒ 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).

ie. per department list all the employees whose salary's are greater than any salary in that department.

Note

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 ~~Exte~~ expensive.

DATA MANIPULATION LANGUAGE

11.1 Insert:

NOTE → SQL → Data manipulation language (DML)
SQL → Data Definetion language (DDL)
SQL → Data Contol language (DCL)

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

inserting multiple rows in relation Movies

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 (Phone_book)>
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 seperating 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 <table_name>;

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>);
Or
Properties
or
Constraint

Example

```
CRETE 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 datatypes:

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

→ 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. It uniquely identifies a row/tuple in a table.

* 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 ~~, alter~~ 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 data base. So that we donot get any error if we donot 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 Data base, different people have different access to different parts of data because same database is being accessed by multiple people in an organization. Hence, a s/w developer, data scientist, admins etc all have different access permission of the database

- s/w developer → may have permission to add, delete, modify etc
- data scientist → can only access the data (read). Cannot modify it
- admins → 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 allows 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 'role1','role2' To 'user1@'localhost', 'user2@'localhost';

→ here these are different roles like admin etc which is given to user1 and user2.

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 databases and all tables of each database

here we are revoking

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

Q1) A relational database contains two tables student and performance as shown below:

Student:

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

Performance:

	Roll_no	Subject Code	Marks
1	1	A	86
2	1	B	95
3	1	C	90
4	2	A	89
5	2	C	92
6	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 from 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)
FROM student S, Performance P
WHERE P.marks > 84
GROUP BY S.student_name;
```

executed 1st: FROM student S, Performance P → This indicates cross product of S and P

executed 2nd: WHERE P.marks > 84 → except 6th row all 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 S
ON B.isbn = S.isbn;
```

Query 2:

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

Query 3:

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

Query 4:

```
SELECT B.isbn, S.copies
FROM B B FULL OUTER JOIN stocks S
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, correct option is the query with FULL OUTER JOIN ie Query 4 (D).

(Q.43) 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 created first

This 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 groupby ~~the~~ based on DeptName ie 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 $\frac{4+3+3+2+1}{5} = \frac{13}{5} = 2.6$.

(84) Consider the following database table named top_scorer:

Top_scorer:

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
Lineker	England	10
T Muller	Germany	10
Rahn	Germany	10

Consider the following query:

```
① { SELECT ta.player FROM top_scorer AS ta
    WHERE ta.goals > ALL (SELECT tb.goals
                          ② { FROM top_scorer AS tb
                              WHERE tb.country = 'Spain')
                              AND ta.goals > ANY (SELECT tc.goals
                                                  ③ { FROM top_scorer AS tc
                                                      WHERE tc.country = 'Germany'));
```

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

⟹ Here we have 3 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 ALL (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 ⇒

tc.goals
16
14
11
10
10

The condition before logical AND is
tq.goals > Any {16, 14, 11, 10} ie tq.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 = 7

(Q5) Consider the following database table named water_schemes:

water_schemes:

Scheme_no	district_name	capacity
1	Ajmer	20
1	Bikaner	10
2	Bikaner	10
3	Bikaner	20
1	Churu	10
2	Churu	20
1	Dungargarh	10

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

```
(1) Q1: With total (name, capacity) as
        Select district_name, sum (capacity)
        From water_schemes
        group by district_name

(2) Q2: with total_avg (capacity) as
        Select avg(capacity)
        from total

(3) Q3: Select name
        from total, total_avg
        Where total.capacity >= total_avg.capacity
```

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

⟹ With is a keyword is introduced by oracle. Newer 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, Q_1 will return ⇒ Total:

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

This name is given by "With"

name	Capacity
Ajmer	20
Bikaner	40
Churu	30

} sum of capacity of each city

NOTE - Using "with" we can access the resultant table as it get saved in memory

The output of Q_2 will be: (This query is using table created by Q_2)

Total_avg:

New table created by Q_2

Capacity
25

→ avg of $\frac{20+40+30+10}{4} = 25$

The output of Q_3 will be: It will use both total and total_avg to get the result. It is performing cartesian product of total and total_avg and returns the tuples whose capacity is greater than average Capacity 25.

hence, only Likanee and Chuvu 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 select columns while selection operation in RA select 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:

Roll_No	Student_Name
1	Raj
2	Rohit
3	Raj

Performance:

Roll_No	Course	Marks
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;
```

} This is equivalent to JOIN

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

(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 (theatre, 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 theatres 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 ~~any~~ even if there is even 1 theatre 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 >= and not >

Hence, correct and feasible option is (A)

(Q9) Given the following statements:

S_1: A foreign key declaration can always be replaced by an equivalent check assertion in SQL.

S_2: Given the table R(a, b, c) where a and b together from the primary key, the following is a valid table definition.

```
CREATE TABLE S (
a Integer,
d Integer,
c Integer,
PRIMARY KEY (d),
FOREIGN KEY (a) References R )
```

Which one of the following statement is CORRECT?

(A) S_1 is TRUE and S_2 is FALSE

(B) Both S_1 and S_2 are TRUE

(C) S_1 is FALSE and S_2 is TRUE

(D) Both S_1 and S_2 are FALSE

→ S_1 : It is not always delete because we do have Cascade option on delete or update.

S_2: 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

Therefore, correct option is (D) 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 sub query.

⇒ Since there is no logical and syntactical error in query therefor, 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.

$\Longrightarrow$ 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:

q
1
1
1
1
2

Option B will return:

q
1
2

Option C will return:

q
1
1
2
2

It will do join

S.q

a
1
2

⋈

R.

q
1
1
2
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	Hasi	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 ALL (empty set) = True

Hence, A.age > ALL( ) is true for all tuples. and it will return A.id for all A.age > all - Therefore, it will return 3 tuples.

(Q 14) Database table by name loan_Records is given below:

Borrower	Bank_manager	loan_Amount
Ramesh	Sundarajan	10000.00
Suresh	Ramgopal	5000.00
Mahesh	Sundarajan	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	Sundarajan
Suresh	Ramgopal
Mahesh	Sundarajan

NATURAL JOIN

T:

Bank manager	Loan_Amou
Sundarajan	10000.00
Ramgopal	5000.00
Sundarajan	7000.00

It will result in distinct 5 tuples :

Borrower	Bank_manager	Loan_Amount
Ramesh	Sundarajan	10k
Ramesh	Sundarajan	7k
Suresh	Ramgopal	5k
Mahesh	sundarjanan	10k
Mahesh	Sundarajan	7k

5 distinct tuples

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