Form of basic SQL query

We will present a number of sample queries using the following table definitions:

Sailors (*sid:* integer, *sname:* string, *rating:* integer, *age:* real)

Boats (*bid:* integer, *bname:* string, *color:* string)

Reserves (*sid:* integer, *bid:* integer, *day:* date)

Queries:

- CREATE TABLE sailors (sid int PRIMARY KEY, sname varchar (50), rating integer ,age int)
- CREATE TABLE boats(bid int PRIMARY KEY,bname varchar(50),color varchar(50));
- CREATE TABLE reserves (sid int,bid int,day date,PRIMARY KEY (sid,bid,day) ,FOREIGN KEY(sid) REFERENCES sailors(sid), FOREIGN KEY(bid) REFERENCES boats(bid));

- INSERT INTO sailors VALUES

  (22,"Dustin",7,45),(29,"Brutus",1,33),(31,"Lubber",8,55),(32,"Andy",8,25),

  (58,"Rusty", 10, 35),(64,"Horatio",7,35),(71,"Zorba",10,16),

  (74,"Horatio", 9, 35),(85,"Art",3,25),(95,"Bob",3,63);
- INSERT INTO boats VALUES

  (101,"Interlake","Blue"), (102,"Interlake","Red"),

  (103,"Clipper","Green"), (104,"Marine","red");
- INSERT INTO reserves VALUES

  (22, 101, 10/10/98), (22, 102, 10/10/98), (22,103,10/8/98), (22,104,10/7/98),

  (31, 102, 11/10/98), (31, 103, 11/6/98), (31,104, 11/12/98), (64,101,9/5/98),

  (64, 102, 9/8/98), (74, 103, 9/8/98);

This section presents the syntax of a simple SQL query and explains its meaning through a *conceptual Evaluation strategy.* A conceptual evaluation strategy is a way to evaluate the query that is intended to be easy to understand rather than efficient.

A DBMS would typically execute a query in a different and more efficient way. The basic form of an SQL query is as follows:

SELECT [DISTINCT] select-list
FROM from-list
WHERE qualification

Example:

SELECT DISTINCT S.sname, S.age FROM Sailors S;

The DISTINCT keyword is optional. It indicates that the table computed as an answer to this query should not contain *duplicates.*

If two or more sailors have the same name and age, the answer still contains just one pair. This query is equivalent to the projection operator of relational algebra.

Example:
SELECT S.sname, S.age FROM Sailors S

Example:

Q: Find all sailors with a rating above 7.

A: SELECT S.sid, S.sname, S.rating, S.age FROM Sailors AS S
WHERE S.rating > 7
Or
SELECT sid, sname, rating, age from sailors where rating > 7

AS is an optional keyword used to assign temporary names to table or column name or both. This is known as creating **Alias in SQL.**

SELECT clause is actually used to do *projection,* whereas *selections* in the relational algebra sense are expressed using the WHERE clause.

Q: ***Find the names of sailors who have reserved boat number 103.***

A: *SELECT S.sname FROM Sailors S, Reserves R WHERE S.sid = R.sid AND R.bid=103*

Q: ***Find the sids of sailors who have reserved a red boat.***

***A*:** SELECT R.sid FROM Boats B, Reserves R WHERE B.bid = R.bid ANDB.color = 'red';

Q: ***Find the names of sailors who have reserved a red boat.***

A: SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red';

Q: ***Find the colors of boats reserved by Lubber.***

A: SELECT B.color FROM Sailors S, Reserves R, Boats B WHERE R.bid = B.bid AND S.sid = R.sid AND S.sname = 'Lubber';

Like Operator:

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.
There are two **wildcards** often used in conjunction with the LIKE operator:

- The percent sign (%) represents zero, one, or multiple characters
- The underscore sign (_) represents one, single character

Sailors

sid	sname	rating	age
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Q: Find the sailors details whose name starts with 'D'.

A: SELECT * from sailors where sname like "D%";

Q: Find the sailors details whose name ends with 'Y'.

A: SELECT * from sailors WHERE sname LIKE "%Y";

***Q:* Find the sailors details whose name contains "bb".**

A: SELECT * from sailors WHERE sname LIKE "%bb%";

***Q:* Find the sailors details whose name contain 'o' in second place.**

A: SELECT * FROM sailors where sname LIKE "_o%";

***Q:* Find the sailors details whose name contain "s" in the third place and name should contain total of five characters.**

A: SELECT * FROM sailors WHERE sname like "__s__";

Q: Find the details of sailors whose names does not starts with "a".

A: SELECT * FROM sailors WHERE sname NOT LIKE "a%";

```
SELECT * FROM sailors WHERE sname  LIKE '%a%'INTERSECT
SELECT * FROM sailors WHERE sname NOT LIKE '%u%'
```

UNION, INTERSECT AND EXCEPT

UNION:

Union is an operator that allows us to combine two or more results from multiple SELECT queries into a single result set. It comes with a default feature that removes the **duplicate** rows from the result set. MySQL always uses the name of the column in the first SELECT statement will be the column names of the result set (output).

Union must follow these basic rules:

- The number and order of the columns should be the same in all tables that you are going to use.
- The data type must be compatible with the corresponding positions of each select query.
- The column name selected in the different SELECT queries must be in the same order.

Syntax:

```
SELECT expression1, expression2, expression_n
FROM tables
[WHERE conditions]
UNION
SELECT expression1, expression2, expression_n
FROM tables
[WHERE conditions];
```

Q: ***Find the sid's and names of sailors who have reserved a red or a green boat.***

A:

```
SELECT S.sid,S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
union
SELECT S2.sid,S2.sname
FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green';
```

INTERSECT:

The INTERSECT operator is used to fetch the records that are in common between two SELECT statements or data sets. If a record exists in one query and not in the other, it will be omitted from the INTERSECT results.

Syntax:

```
SELECT expression1, expression2, ... expression_n
FROM tables
[WHERE conditions]
 INTERSECT
SELECT expression1, expression2, ... expression_n
FROM tables
[WHERE conditions];
```

Intersection must follow these basic rules:

- The number and order of the columns should be the same in all tables that you are going to use.
- The data type must be compatible with the corresponding positions of each select query.
- The column name selected in the different SELECT queries must be in the same order.

Q: ***Find the sid's and names of sailors who have reserved a red and a green boat.***

A:

```
SELECT S.sid,S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
INTERSECT
SELECT S2.sid,S2.sname
FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green';
```

EXCEPT:

The EXCEPT clause in SQL is widely used to filter records from more than one table. This statement first combines the two SELECT statements and returns records from the first SELECT query, which isn’t present in the second SELECT query's result. In other words, it retrieves all rows from the first SELECT query while deleting redundant rows from the second.

This statement behaves the same as the minus (set difference) operator does in mathematics.

Rules for SQL EXCEPT

We should consider the following rules before using the EXCEPT statement in SQL:

- In all SELECT statements, the number of columns and orders in the tables must be the same.
- The corresponding column's data types should be either the same or compatible.
- The fields in the respective columns of two SELECT statements cannot be the same.

Syntax:

```
SELECT column_lists from table_name1
EXCEPT
SELECT column_lists from table_name2;
```

Q: Find the sids of all sailors who have reserved red boats but not green boats.

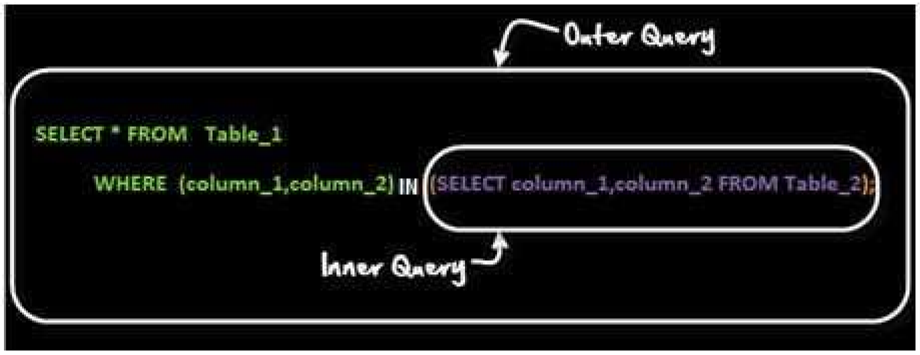
A:

```
SELECT S.sid
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
MINUS // minus in Oracle, Except in SQL server
SELECT S2.sid
FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green';
```

Nested Queries

A nested query is a query that has another query embedded within it; the embedded query is called a sub query. A sub query is known as the inner query, and the query that contains sub query is known as the outer query.

Syntax:



```
SELECT * FROM  Table_1
      WHERE  (column_1,column_2) IN (SELECT column_1,column_2 FROM Table_2);
```

As per the execution process of sub query, it again classified in two ways.

1. **Non- Correlated sub queries.**

   In this sub queries, First inner query is executed and later Outer query will execute. i.e Outer query always depends on the result of inner query.

2. **Correlated Queries.**

   In correlated queries, First Outer query is executed and later Inner query will execute. i.e Inner query always depends on the result of outer query.

Example for Non – Correlated Queries:

Q: Find the names of sailors who have reserved boat 103.

A: SELECT S.sname from sailors S WHERE S.sid IN (SELECT R.sid FROM reserves R WHERE R.bid=103);

SQL IN condition used to allow multiple value in a WHERE clause condition.
SQL IN condition you can use when you need to use multiple OR condition.

(Q) Find the names of sailors who have reserved a red boat.

A: SELECT S.sname from sailors S WHERE S.sid IN

```
(SELECT R.sid FROM reserves R where R.bid IN
(SELECT B.bid FROM boats B WHERE B.color="red"));
```

SQL NOT IN condition used to exclude the defined multiple value in a WHERE clause condition.

Correlated Sub Queries:

Correlated sub queries are the one in which **inner query or sub query reference outer query**. Outer query needs to be executed before inner query. One of the most common examples of correlated sub query is using keywords exits and not exits.

Example:

(Q) Find the names of sailors who have reserved boat number 103

```
SELECT S.sname FROM Sailors S WHERE
EXISTS (SELECT * FROM Reserves R WHERE R.bid = 103 AND R.sid = S.sid)
```

Aggregation Operators:

SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value. It is also used to summarize the data.

SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.
2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.
3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.
4. MAX (A): The maximum value in the A column.
5. MIN (A): The minimum value in the A column.

Example:

```
CREATE TABLE employees(id int,name char(50),dob date,gender char(5),sal
int,dept char(10),collegename char(15),age int);
DESC employees;
INSERT INTO employees VALUES
(1,"Srikanth",'1990-05-05','M',25000,'cse','kits',25),
(2,"Kamal",'1992-02-02','M',65000,'ece','kitsw',29),
(3,"kavitha",'1993-01-01','F',45000,'cse','kits',30),
(4,"Karun",'1988-06-04','M',25000,'cse','kits',31),
(5,"Gouthami",'1980-02-09','F',10000,'civil','kits',29),
(6,"Narendher",'1985-09-18','M',30000,'civil','kits',35),
(7,"Mahesh",'1990-03-06','M',8000,'cse','kitsw',45);
(8,    NULL,"1992-06-09",'F',65000,"ece",'kitsw',46),
(9,"Sravanthi","1991-07-04",'F',46000,"CSE",'kitsw',29),
(1,"Neeraja","1991-07-04",'F',50000,"CSE",'kitsw',26);
```

Queries:

```
SELECT COUNT(*) from employees; // Output=10
SELECT COUNT(name) FROM employees; // Output=9
SELECT COUNT(id) FROM employees; // Output=10
SELECT COUNT(DISTINCT id) FROM employees; // Output=9
SELECT COUNT(*) FROM employees WHERE collegename ="kitsw";
                                                        // Output=5
SELECT MAX(sal) FROM employees; // Output=65000
SELECT MAX(sal) as Maximum_salary FROM employees; // Output=65000
SELECT MAX(sal) FROM employees WHERE collegename="kits";
                                                        // Output=65000
 SELECT MIN(sal) FROM employees; // Output=8000
 SELECT MIN(sal) FROM employees WHERE collegename='kits';
                                                        // Output=8000
 SELECT SUM(sal) FROM employees;  // Output=369000
 SELECT sum(sal) as total_salary FROM employees; // Output=369000
 SELECT AVG(sal) FROM employees;  // Output=36900
```

Order by Clause:

Order by clause is used sort the rows either in ascending order or descending order (default is ascending)

Example:

```
SELECT * from employees ORDER BY sal ASC;
SELECT * FROM employees ORDER BY sal DESC;
```

Group By clause:

The GROUP BY Statement in SQL is used to arrange identical data into groups with the help of some functions. i.e if a particular column has same values in different rows then it will arrange these rows in a group.

Important Points:

- GROUP BY clause is used with the SELECT statement.
- In the query, GROUP BY clause is placed after the WHERE clause.
- In the query, GROUP BY clause is placed before ORDER BY clause if used any.

Syntax:

```
SELECT column1, function_name(column2)
FROM table_name
WHERE condition
GROUP BY column1, column2
ORDER BY column1, column2;
```

function_name: Name of the function used for example, SUM() , AVG().
table_name: Name of the table.
condition: Condition used.

Examples:

Q: Display the Number of employees in each department.

A: SELECT dept,COUNT(*) FROM employees GROUP BY dept;
B: SELECT dept,COUNT(*) AS No_of_employees FROM employees GROUP BYdept;

Q: What is the highest salary in each department?

A: SELECT dept,MAX(sal) from employees GROUP BY dept;

Q: What is the total salary each department is paying?

A: SELECT dept,SUM(sal) AS total_amount_paying from employees GROUP BY dept;

Q: Display the Number of male and female faculty.

A: SELECT gender,COUNT(*) FROM employees GROUP BY gender;

Q: What is the highest and lowest salary where each dept is paying.

A: SELECT dept,MAX(sal),MIN(sal) FROM employees GROUP BY dept;

Q: Display the number of male and female faculty from each Department.

A: SELECT dept,gender,COUNT(*) FROM employees GROUP BY dept, gender;

Q: Display the number of male and female's faculty from each college.

A: SELECT collegename,gender,COUNT(*) from employees GROUP BY collegename , gender;

Having Clause:

The HAVING clause is like WHERE but operates on grouped records returned by a GROUP BY. HAVING applies to summarized group records, whereas WHERE applies to individual records. Only the groups that meet the HAVING criteria will be returned.

HAVING requires that a GROUP BY clause is present.
Both WHERE and HAVING can be used in the same query at the same time.

Syntax:

```
SELECT column-names
FROM  table-name
WHERE  condition
GROUP BY column-names
HAVING condition;
```

Example:

Query: SELECT dept,COUNT(*) FROM employees GROUP BY dept HAVING COUNT(*)>2;

Query: SELECT dept,MAX(age) FROM employees GROUP BY dept HAVING

MAX(sal)>30;

Query: SELECT dept,sum(sal) FROM employees GROUP BY dept HAVING SUM(sal) > 50000;

NULL

We have assumed that column values in a row are always known. In practice column values can be unknown. For example, when a sailor, say Dan, joins a yacht club, he may not yet have a rating assigned. Since the definition for the Sailors table has a rating column, what row should we insert for Dan? What is needed here is a special value that denotes unknown. Suppose the Sailor table definition was modified to include a maiden-name column. However, only married women who take their husband's last name have a maiden name. For women who do not take their husband's name and for men, the maiden-name column is inapplicable. Again, what value do we include in this column for the row representing Dan? SQL provides a special column value called null to use in such situations. We use null when the column value is either Unknown or inapplicable. Using our Sailor table definition, we might enter the row (98. Dan, null, 39) to represent Dan. The presence of null values complicates many issues, and we consider the impact of null values on SQL in this section.

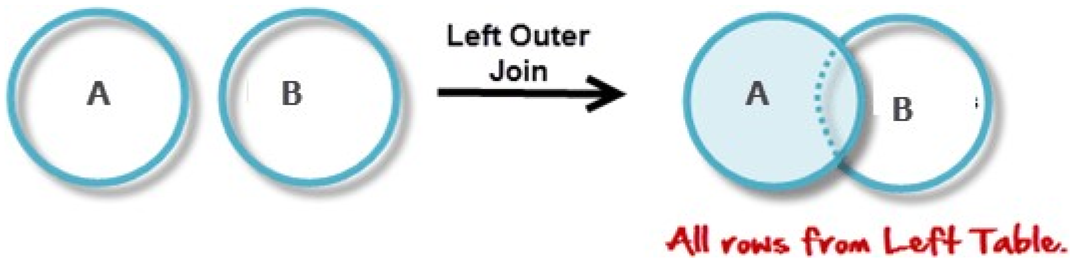
JOINS

OUTER JOIN:

In an outer join, along with tuples that satisfy the matching criteria, we also include some or all tuples that do not match the criteria.

Left Outer Join (A ⟕ B)

In the left outer join, operation allows keeping all tuple in the left relation. However, if there is no matching tuple is found in right relation, then the attributes of right relation in the join result are filled with null values.



Example:

Student

id	Name
101	Raju
102	Rishi
103	Karthik

Marks

id	marks
102	92
103	99

Student ⟕ Marks

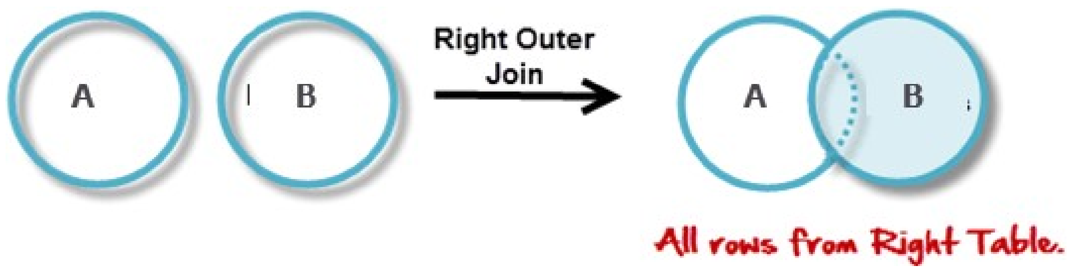
id	Name	id	marks
101	Raju	NULL	NULL
102	Rishi	102	92
103	Karthik	103	99

- CREATE TABLE student (id int, name char(50));
- INSERT INTO student VALUES (101,"raju"),(102,"rishi"),(103,"karthik");
- CREATE TABLE marks (id int,marks int);
- INSERT INTO marks values(102,92),(103,99);

SELECT * from student LEFT OUTER JOIN marks on student.id=marks.id;

Right Outer Join: (A ⟖ B)

In the right outer join, operation allows keeping all tuple in the right relation. However, if there is no matching tuple is found in the left relation, then the attributes of the left relation in the join result are filled with null values.


A
B
Right Outer Join
A
B
All rows from Right Table.

Example:

Student

Id	Name
101	Raju
102	Rishi
103	Karthik

Marks

id	marks
102	92
103	99

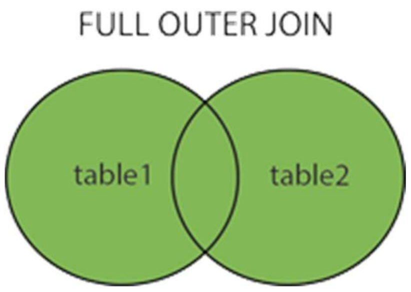
Student ⟖ Marks

id	Name	id	marks
102	Rishi	102	92
103	Karthik	103	99

❖ **SELECT * from student RIGHT OUTER JOIN marks on student.id=marks.id**

Full Outer Join / Full Join: (A ⟗ B)

In a full outer join, all tuples from both relations are included in the result, irrespective of the matching condition.



Student

Id	Name
101	Raju
102	Rishi
103	Karthik

Marks

id	marks
102	92
103	99

Student ⟗ Marks

id	Name	id	marks
101	Raju	NULL	NULL
102	Rishi	102	92
103	Karthik	103	99
102	Rishi	102	92

103	Karthik	103	99

- SELECT * from student LEFT OUTER JOIN marks on student.id=marks.idUNION ALL
  SELECT * from student RIGHT OUTER JOIN marks on student.id=marks.id

SELECT * from student FULL JOIN marks on student.id=marks.id