Unit- 6: Manipulating and Querying Data

Adding Data with Insert Statement

- ➔ To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted.
- ➔ Obviously, the attribute values for inserted tuples must be members of the corresponding attribute's domain.
- ➔ Similarly, tuples inserted must have the correct number of attributes.

  Syntax:

  INSERT INTO table_name VALUES(val1, val2, …, valN);

  Or

  INSERT INTO table_name(col1,col2,col3,…colN)

  VALUES(val1,val2,val3,…,valN);

  - ☞ In this first case, the values are specified in the order in which the corresponding attributes are listed in the relation schema.
  - ☞ For the benefit of users who may not remember the order of the attributes, SQL allows the attributes to be specified as part of the insert statement as in second case

Examples:

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Pokhara', 2000.00 );
```

```
INSERT INTO CUSTOMERS
VALUES (7, 'Hari', 24, 'Kathmandu', 10000.00 );
```

Retrieving data with SELECT statement and FROM Clause and filter data with WHERE clause

SELECT Statement

The **SELECT statement** is used to select data from a database. The data returned is stored in a result table, called the result-set.

Syntax:

```
SELECT column1, column2, ...
FROM table_name;
```

Here, column1, column2, ... are the field names of the table you want to select data from.

If you want to select all the fields available in the table, use the following syntax:

```
SELECT * FROM table_name;
```

Example:

Consider a following table

Sailors

Sid	sname	Rating	Age
1	Ajaya	12	33
2	Robin	11	43
3	Ganga	32	28
4	Manoj	9	31
7	Rahul	7	22
9	Sanjaya	9	42
11	Raju	4	19

Now we want to select the content of the columns named "sname" and "age" from the table Sailors. We use the following SELECT statement:

SELECT sname, age

FROM Sailors;

The result-set will look like this:

Sname	Age
Ajaya	33
Robin	43
Ganga	28
Manoj	31
Rahul	22
Sanjaya	42
Raju	19

Now we want to select all the columns from the ―Sailors " table. We use the following SELECT statement:

*SELECT * FROM Sailors;*

The result-set will look like this:

Sid	Sname	Rating	Age
1	Ajaya	12	33
2	Robin	11	43
3	Ganga	32	28
4	Manoj	9	31
7	Rahul	7	22
9	Sanjaya	9	42
11	Raju	4	19

The SQL SELECT DISTINCT Statement:

- In a table, some of the columns may contain duplicate values. This is not a problem; however, sometimes you will want to list only the different (distinct) values in a table. The **DISTINCT** keyword can be used to return only distinct (different) values.
- SQL SELECT DISTINCT Syntax:

SELECT DISTINCT column_name(s) FROM table_name ;

Example: Now we want to select only the distinct values from the column named "bname" from

Following the table —Boats.

Boats

Bid	Bname	Color
11	Marine	Red
24	Clipper	Blue
33	Wooden	Black
41	Marine	Green

We use the following SELECT statement:

SELECT DISTINCT bname FROM Boats ;

The result-set will look like this:

Bname
Marine
Clipper
Wooden

- The **select** clause can contain arithmetic expressions involving the operation, +, –, □, and /, and operating on constants or attributes of tuples.
  - The query:

    select *ID, name, salary/12*
    from *instructor*

would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.

- Can rename "s*alary/12"* using the **as** clause:

  select *ID, name, salary/12* **as** *monthly_salary*

The where Clause

The WHERE clause is used to filter records.

The WHERE clause is used to extract only those records that fulfill a specified condition

The where clause specifies conditions that the result must satisfy

- SQL WHERE Syntax:

SELECT column_name(s) FROM table_name WHERE column_name operator value

- SQL allows the use of the logical connectives **and, or**, and **not**
- The operands of the logical connectives can be expressions involving the comparison operators **<, <=, >, >=, =,** and **<>.**
- Comparisons can be applied to results of arithmetic expressions

Example:

Now we want to select only those Sailors whose age is less than 30 from the table Sailors above.

We use the following SELECT statement:

*SELECT * FROM Sailors*

WHERE age < 30;

The result-set will look like this:

Sid	sname	Rating	Age
3	Ganga	32	28
7	Rahul	7	22
11	Raju	4	19

Quotes around Text Fields

- ♣ SQL uses single quotes around text values (most database systems will also accept double quotes).
- ♣ Although, numeric values should not be enclosed in quotes.

For text values:

This is correct:

```
SELECT * FROM Sailors
WHERE sname='Ajaya';
```

This is wrong:

```
SELECT * FROM Sailors
WHERE sname=Ajaya;
```

For Numeric values:

This is correct:

```
SELECT * FROM Sailors
WHERE age=32;
```

This is wrong:

```
SELECT * FROM Sailors
WHERE age='32';
```

Operators Allowed in the WHERE Clause

Operator	Description
=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern
IN	If you know the exact value you want to return for at least one of the columns.
AND	And
OR	Or

The AND & OR Operators

- ♣ The AND & OR operators are used to filter records based on more than one condition.
- ♣ The AND operator displays a record if both the first condition and the second condition is true.
- ♣ The OR operator displays a record if either the first condition or the second condition is true.

AND Operator Example:

Suppose we want to select only the Sailors with the name equal to "Ajaya" AND the age equal to 33: We use the following SELECT statement:

*SELECT * FROM Sailors*

WHERE sname='Ajaya' AND age=33;

The result-set will look like this:

Sid	sname	rating	Age
1	Ajaya	12	33

Example 2: Find the sids of sailors who have reserved a red boat.

Reserves

Sid	Bid	Day
1	24	2068-08-11
1	11	2068-08-11
1	41	2068-08-22
2	33	2068-11-08
2	11	2068-08-19
11	41	2068-09-23
9	24	2068-08-10
9	11	2069-08-15
9	33	2068-05-21

```
SELECT R.sid

FROM Boats B, Reserves R

WHERE B.bid = R.bid AND B.color = `red'
```

The result-set will look like this:

Sid
1
2
9

OR Operator Example:
Now we want to select only the Sailors with the first name equal to "Rahul" OR the rating equal to 9:

We use the following SELECT statement:

*SELECT * FROM Sailors*

WHERE sname='Rahul' OR rating=9;

The result-set will look like this

Sid	Sname	Rating	Age
4	Manoj	9	31
7	Rahul	7	22
9	Sanjaya	9	42

Combining AND & OR:

We can also combine AND and OR (use parenthesis to form complex expressions).

Now we want to select only the Sailors of rating equal to 9 AND the age equal to 31 OR to 42:

We use the following SELECT statement:

*SELECT * FROM Sailors*

WHERE rating=9 AND (age=31 OR age=42);

The result-set will look like this:

Sid	Sname	Rating	Age
4	Manoj	9	31
9	Sanjaya	9	42

The SQL BETWEEN Operator

- The **BETWEEN** operator is used to select values within a range. The values can be numbers, text, or dates.

♣ Syntax:

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

Example:

The following SQL statement selects all Sailors with age BETWEEN 20 and 40:

*SELECT * FROM Sailors*

WHERE age BETWEEN 20 AND 40;

The result-set will look like this:

Sid	Sname	Rating	Age
7	Rahul	7	22
4	Manoj	9	31
3	Ganga	32	28
1	Ajaya	12	33

NOT BETWEEN Operator Example:

To display the Sailors outside the range of the previous example, use **NOT BETWEEN**:

*SELECT * FROM Sailors*

WHERE age NOT BETWEEN 20 AND 40;

The result-set will look like this:

Sid	Sname	Rating	Age
9	Sanjaya	9	42
2	Robin	11	43
11	Raju	4	19

SQL IN Operator

♣ The IN operator allows us to specify multiple values in a WHERE clause.

♣ SQL IN Syntax:

SELECT column_name(s) FROM table_name

WHERE column_name IN (value1, value2...);

IN Operator Example:

The following SQL statement selects all Sailors with a rating of 9 or 11:

*SELECT * FROM Sailors*

WHERE Rating IN (9, 11);

The result-set will look like this

Sid	sname	Rating	Age
2	Robin	11	43
4	Manoj	9	31
9	Sanjaya	9	42

Its equivalent query by using OR operator is as below:

*SELECT * FROM Sailors WHERE Rating=9 OR Rating=11;*

NOT IN Operator Example:

The following SQL statement selects all Sailors with age not 11 or 9:

*SELECT *FROM Sailors*

WHERE rating NOT IN (11, 9);

The result-set will look like this:

Sid	sname	Rating	age
1	Ajaya	12	33
3	Ganga	32	28
7	Rahul	7	22
11	Raju	4	19

The from Clause

- The from clause lists the relations (tables) involved in the query

  Select* from instructor, teaches

  Generates every possible instructor – teaches pair, with all attributes from both relations.

Examples

- Find the names of all instructors who have taught some course and the course_id
  - **select** *name, course_id*
    from *instructor , teaches*
    where *instructor.ID = teaches.ID*
- Find the names of all instructors in the Art department who have taught some course and the course_id
  - **select** *name, course_id*
    from *instructor , teaches*
    where *instructor.ID = teaches.ID* ***and*** *instructor. dept_name =* 'Art'

Order and Grouping data with ORDER BY and GROUP BY Clause

ORDER BY

- The ORDER BY keyword is used to sort the result-set in ascending or descending order.
- The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

  Syntax:

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

Consider the following relation “Customers”

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
3	kaushik	23	Kota	2000.00
2	Khilan	25	Delhi	1500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00
1	Ramesh	32	Ahmedabad	2000.00

SELECT * FROM CUSTOMERS

ORDER BY NAME DESC;

This would produce the following result −

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
7	Muffy	24	Indore	10000.00
6	Komal	22	MP	4500.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
5	Hardik	27	Bhopal	8500.00
4	Chaitali	25	Mumbai	6500.00

Aggregate Functions

- **Aggregate functions** are functions that take a collection of values as input and return a single value.
- Some useful aggregate functions are
    - SUM() - Returns the sum (total)
    - AVG() - Returns the average value
    - COUNT() - Returns the number of rows

- MAX() - Returns the largest value
- MIN() - Returns the smallest value

- The input to **sum()** and **avg**() must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.

Example 1: find sum of rating of all sailors.

SELECT SUM (rating)

FROM Sailors;

The result-set will look like this:

Sum(rating)
85

Example 2: find average age of all sailors.

SELECT AVG (age)

FROM Sailors;

The result-set will look like this:

Avg(age)
29.7143

Example 3: find average age of all sailors with a rating of 9.

SELECT AVG (age)

FROM Sailors

WHERE rating=9;

Avg(age)
31.5000

Example 4: count number of sailors

SELECT COUNT(*)

FROM Sailors;

The result-set will look like this:

count(*)
7

- SQL does not allow the use of distinct with count (*). It is legal to use distinct with max and min, even though the result does not change

GROUP BY Clause

There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this wish in SQL using the group by clause. The attribute or attributes given in the group by clause are used to form groups. Tuples with the same value on all attributes in the group by clause are placed in one group.

The SQL GROUP BY clause is use to divide the rows in a table into groups. The GROUP BY statement is used along with the SQL aggregate functions. In GROUP BY clause, the tuples with same values are placed in one group.

Example: Find the average salary in each department

select dept name, avg (salary) as avg_salary

from instructor

group by dept_name;

Another Example: Find the age of the youngest sailor for each rating level.

SELECT rating, MIN(age)

FROM Sailors

GROUP BY rating;

The result-set will look like this

Rating	Min(age)
4	19
7	22
9	31
11	43
12	33
32	28

This table display the minimum age of each group according to their rating.

When an SQL query uses grouping, it is important to ensure that the only attributes that appear in the select statement without being aggregated are those that are present in the group by clause. In other words, any attribute that is not present in the group by clause must appear only inside an aggregate function if it appears in the select clause, otherwise the query is treated as erroneous

/ erroneous query */*

select dept_name, ID, avg (salary)

from instructor

group by dept_name;

SQL HAVING Clause:

The SQL HAVING clause allows us to specify conditions on the rows for each group. It is used instead of the WHERE clause when Aggregate Functions are used. HAVING clause should follow the GROUP BY clause if we are using it.

It is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those departments where the average salary of the instructors is more than $42,000. This condition does not apply to a single tuple; rather, it applies to each group constructed

by the group by clause. To express such a query, we use the having clause of SQL. SQL applies predicates in the having clause after groups have been formed, so aggregate functions may be used

We express this query in SQL as follows:

select dept_ name, avg (salary) as avg_salary

from instructor

group by dept_name

having avg (salary) > 42000;

- As was the case for the **select** clause, any attribute that is present in the **having** clause without being aggregated must appear in the **group by** clause, otherwise the query is treated as erroneous.

The meaning of a query containing aggregation, group by, or having clauses is defined by the following sequence of operations

- As was the case for queries without aggregation, the **from** clause is first evaluated to get a relation
- If a **where** clause is present, the predicate in the **where** clause is applied on the result relation of the from clause
- Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause if it is present. If the **group by** clause is absent, the entire set of tuples satisfying the **where** predicate is treated as being in one group
- The **having** clause, if it is present, is applied to each group; the groups that do not satisfy the **having** clause predicate are removed
- The **select** clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.

Null Values

SQL allows the use of NULL values to indicate absence of information about the value of an attribute. It has a special meaning in the database- the value of the column is not currently known but its value may be known at a later time.

- It is possible for tuples to have a null value, denoted by **null**, for some of their attributes
- **null** signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving **null** is **null**
    - Example: 5 + **null** returns **null**

A special comparison operator IS NULL is used to test a column value for NULL. It has following general format:

column_name IS [NOT] NULL;

This comparison operator return *true* if column_name contains NULL, otherwise return *false.* The optional NOT reverses the result.

Following syntax is illegal in SQL:

WHERE attribute=NULL;

Example: Find all instructors whose salary is null.

select *name*
from *instructor*
where *salary* **is null**

String Operations

SQL specifies strings by enclosing them in single quotes, for example 'Pokhara'. The most commonly used operation on strings is pattern matching. It uses the operator LIKE. We describe the patterns by using two special characters:

- **Percent (%):** The % character matches any substring, even the empty string.

- **Underscore (_):** The underscore stands for exactly one character. It matches any character.

To illustrate pattern matching, we consider the following examples:

- 'A_Z': All string that starts with 'A', another character and end with 'Z'. For example, 'ABZ' and 'A2Z' both satisfy this condition but 'ABHZ' does not because between A and Z there are two characters are present instead of one.
- 'ABC%': All strings that start with 'ABC'.
- '%ABC': All strings that ends with 'ABC'.
- '%AN%': All strings that contains the pattern 'AN' anywhere. For example, 'ANGELS' ,'SAN', 'FRANCISCO' etc.
- '_ _ _': matches any strings of exactly three characters.
- '_ _ _%': matches any strings of at least three characters

SQL expresses patterns by using the **like** comparison operator.

Consider the query

"Find the names of all departments whose building name includes the substring 'Watson'."

This query can be written as:

select dept name

from department

where building like '%Watson%';

Another Example

*SELECT * FROM Sailors WHERE sname LIKE '%ya';*

This SQL statement will match any Sailors first names that end with 'ya'.
The result-set will look like this

Sid	sname	Rating	age
1	Ajaya	12	33
9	Sanjaya	9	42

For patterns to include the special pattern characters (that is,%and), SQL allows the specification of an **escape character**. The **escape character** is used immediately before a special pattern character to indicate that the special pattern character is to be treated like a normal character. We define the escape character for a like comparison using the escape keyword. To illustrate, consider the following patterns, which use a backslash (\) as the escape character:

- like 'ab\%cd%' escape '\' matches all strings beginning with "ab%cd".
- like 'ab\\cd%' escape '\' matches all strings beginning with "ab\cd".

- SQL supports a variety of string operations such as
  - concatenation (using "||")
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc

    (There are different functions for different string operations. Explore yourself for detail)

Nested Subqueries

- A Subquery or Inner query or a Nested query is a query within another SQL query
- A subquery is a select-fromwhere expression that is nested within another query.

- A subquery is a query within another query. The outer query is known as the main query, and the inner query is known as a subquery. The result of sub query is used by the main query (outer query).
- A common use of subqueries is to perform tests for set membership, make set comparisons, and determine set cardinality, by nesting subqueries in the where clause.
- We can place the sub-query in a number of SQL clauses including:
    - The WHERE clause
    - The HAVING clause
    - The FROM clause
- Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements.
- Subqueries are placed on the right side of the comparison operator.
- Subqueries must be enclosed within parentheses.
- Inner query is executed before outer query.
- Subqueries that can return only one or zero rows to the outer query are called single-row subqueries. Comparison operators (=, <>, >, <, <=) can be used with a single subquery.
- Subqueries that can return more than one row to the outer query are called multiple-row subqueries. IN, ANY(or SOME) , or ALL operators are used in outer query to handle result set produced by a nulti row subquery.

Set Membership

SQL allows testing tuples for membership in a relation. The IN connective tests for set membership, where the set is a collection of values produced by a select clause. The NOT IN connective tests for the absence of set membership

Example1: Find the names of sailors who have reserved boat 41.

SELECT sname FROM Sailors

WHERE sid IN (SELECT sid

FROM Reserves

WHERE bid=41);

Sname
Ajaya
Raju

Example 2: Find the names of sailors who have reserved a red boat.

SELECT sname

FROM sailors

WHERE sid IN (SELECT sid

FROM Reservs

WHERE bid IN (SELECT bid

FROM Boats

WHERE color='Red'));

sname
Ajaya
Robin
Sanjaya

Example 3: Find the names of sailors who have not reserved a red boat.

SELECT sname

FROM sailors

WHERE sid NOT IN (SELECT sid

FROM Reservs WHERE bid IN (SELECT bid

FROM Boats

WHERE color='Red'));

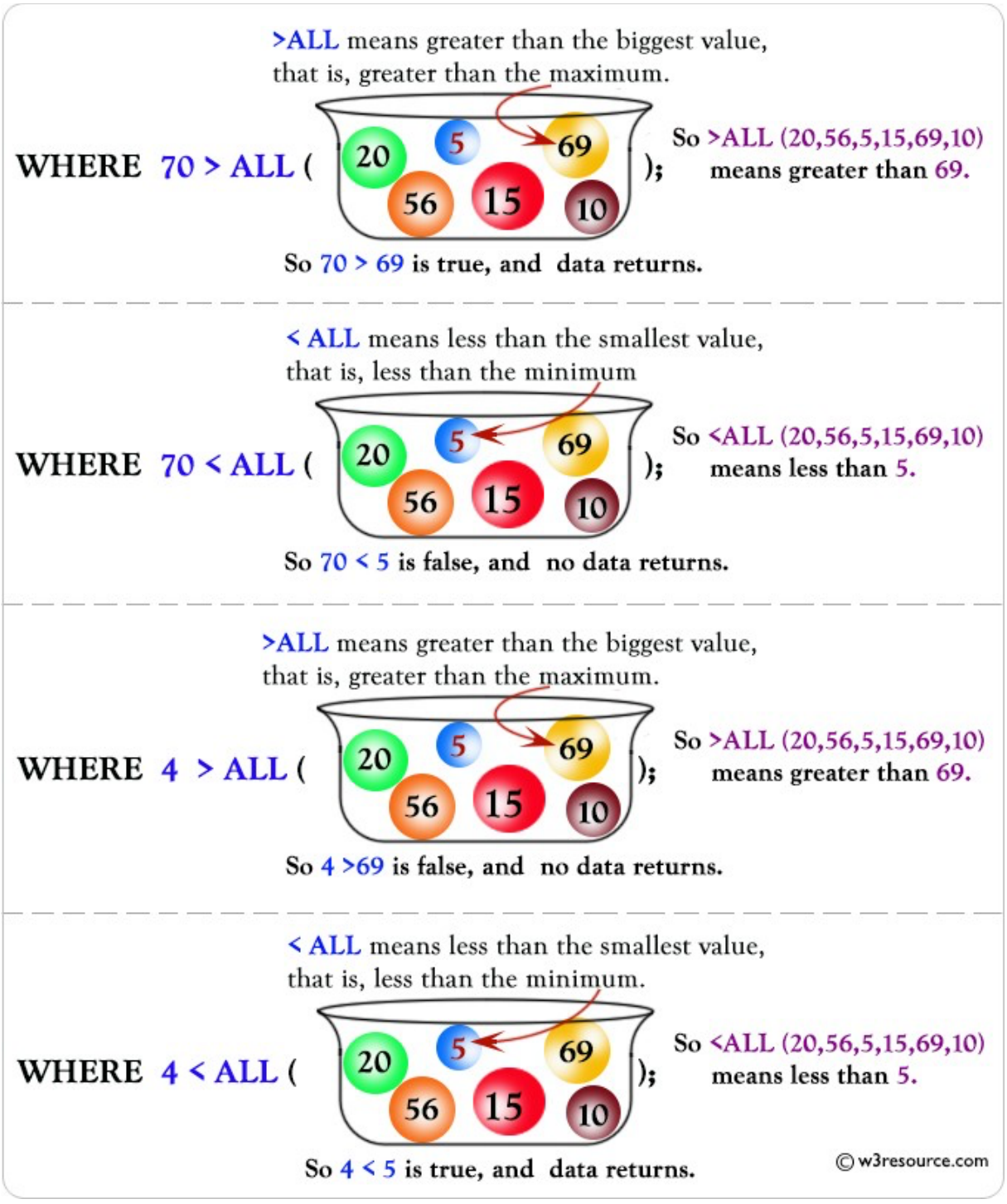
Set Comparison

The comparison operators are used to compare sets in nested sub-query. SQL allows following set comparisons: **< SOME, <= SOME, > SOME, >= SOME, = SOME, < > SOME <ALL, <=ALL, >ALL, >=ALL, = ALL, < >ALL** .The keyword **ANY** is synonymous to **SOME** in SQL.

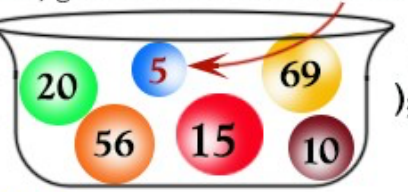
- The **ALL** operator returns TRUE iff all of the subqueries values meet the condition. The **ALL must be preceded by comparison operators** and evaluates true if all of the subqueries values meet the condition.
- **ANY (or SOME)** return true if any of the subqueries values meet the condition. **ANY must be preceded by comparison operators**.

NOTE : **= SOME** is identical to **IN** , whereas **<> SOME** is not the same as **NOT IN**

<> ALL is identical to **NOT IN**, whereas **= ALL** is not the same as **IN**


>ALL means greater than the biggest value,
that is, greater than the maximum.
WHERE 70 > ALL (20 5 69 56 15 10);
So >ALL (20,56,5,15,69,10) means greater than 69.
So 70 > 69 is true, and data returns.
< ALL means less than the smallest value,
that is, less than the minimum
WHERE 70 < ALL (20 5 69 56 15 10);
So <ALL (20,56,5,15,69,10) means less than 5.
So 70 < 5 is false, and no data returns.
>ALL means greater than the biggest value,
that is, greater than the maximum.
WHERE 4 > ALL (20 5 69 56 15 10);
So >ALL (20,56,5,15,69,10) means greater than 69.
So 4 >69 is false, and no data returns.
< ALL means less than the smallest value,
that is, less than the minimum.
WHERE 4 < ALL (20 5 69 56 15 10);
So <ALL (20,56,5,15,69,10) means less than 5.
So 4 < 5 is true, and data returns.
© w3resource.com

>ANY means greater than at least one value,
that is, greater than the minimum.

WHERE 70 > ANY (

); So >ANY (20,56,5,15,69,10) means greater than 5.

So 70 >5 is true, and data returns.

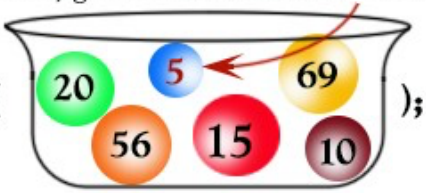
< ANY means less than at least one value,
that is, less than the maximum.

WHERE 70 < ANY (

); So <ANY (20,56,5,15,69,10) means less than 69.

So 70 < 69 is false, and no data returns.

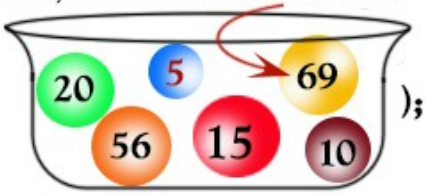
>ANY means greater than at least one value,
that is, greater than the minimum.

WHERE 4 > ANY (

); So >ANY (20,56,5,15,69,10) means greater than 5.

So 4 >5 is false, and no data returns.

< ANY means less than at least one value,
that is, less than the maximum.

WHERE 4 < ANY (

); So <ANY (20,56,5,15,69,10) means less than 69.

So 4 < 69 is true, and data returns.

S4

Sid	Sname	Rating	Age
1	Ajaya	12	33
2	Robin	11	43
3	Ganga	32	28
4	Manoj	9	31
7	Rahul	7	22
9	Sanjaya	9	42
11	Raju	4	19
8	Rahul	6	76

Find the id and names of sailors whose rating is better than some sailor called "Rahul".

```
SELECT sid, sname
FROM S4
WHERE rating >ANY (SELECT rating
                   FROM S4
                   WHERE sname='Rahul');
```

Sid	Sname
1	Ajaya
2	Robin
3	Ganga
4	Manoj
7	Rahul
9	Sanjaya

Find the id and name of sailor with heghest rating.

SELECT sid, sname

FROM S4

WHERE rating >=ALL (SELECT rating

FROM S4);

Sid	Sname
3	Ganga

Test for Empty Relations

SQL includes a feature for testing whether a subquery has any tuples in its result. The EXISTS construct returns the value true if the argument subquery is nonempty.

- The EXISTS onstruct return true if subquery returns value

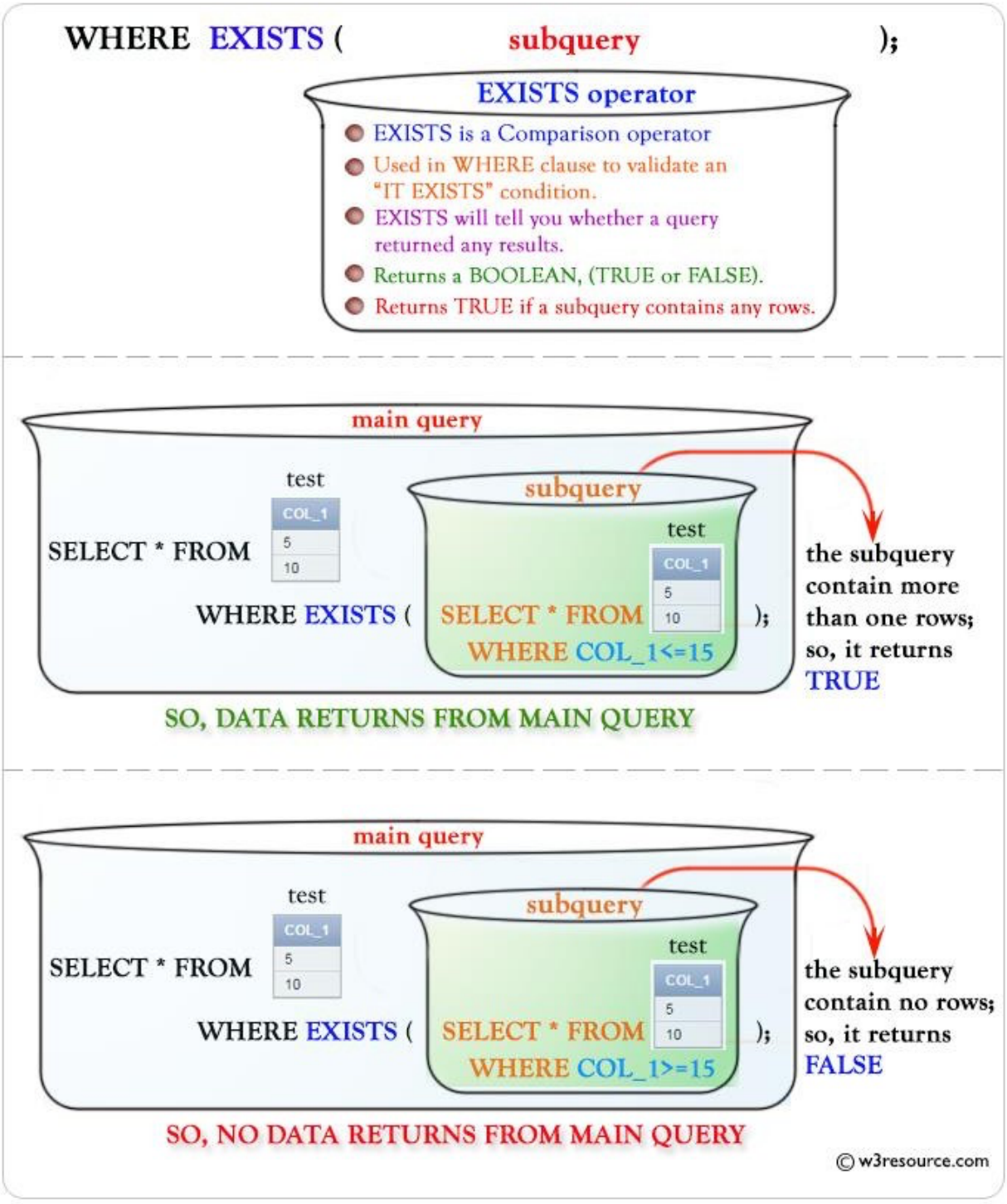
We can test for the nonexistence of tuples in a subquery by using the NOT EXISTS construct

CUSTOMER

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

```
SELECT AGE FROM CUSTOMERS
WHERE EXISTS (SELECT AGE
              FROM CUSTOMERS
              WHERE SALARY > 6500);
```

```
+-----+
| AGE |
+-----+
|  32 |
|  25 |
|  23 |
|  25 |
|  27 |
|  22 |
|  24 |
+-----+
```


WHERE EXISTS (subquery);
EXISTS operator
EXISTS is a Comparison operator
Used in WHERE clause to validate an "IT EXISTS" condition.
EXISTS will tell you whether a query returned any results.
Returns a BOOLEAN, (TRUE or FALSE).
Returns TRUE if a subquery contains any rows.
main query
test
COL_1
5
10
SELECT * FROM
WHERE EXISTS (
subquery
test
COL_1
5
10
SELECT * FROM
WHERE COL_1<=15
);
the subquery contain more than one rows; so, it returns TRUE
SO, DATA RETURNS FROM MAIN QUERY
main query
test
COL_1
5
10
SELECT * FROM
WHERE EXISTS (
subquery
test
COL_1
5
10
SELECT * FROM
WHERE COL_1>=15
);
the subquery contain no rows; so, it returns FALSE
SO, NO DATA RETURNS FROM MAIN QUERY
© w3resource.com

Test for the Absence of Duplicate Tuples

SQL includes a boolean function for testing whether a subquery has duplicate tuples in its result. The UNIQUE construct returns the value true if the argument subquery contains no duplicate tuples.

SQL has a feature for testing whether the subquery has any duplicate tuples in tits results.

The UNIQUE construct returns true if a subquery contains no duplicate records in its result.

The NOT UNIQUE construct returns true if a subquery contains duplicate records in its result.

Example: "Find all courses that were offered at most once in 2009"

```
select T.course_id
  from course as T
  where unique ( select R.course_id
                   from section as R
                   where T.course_id= R.course_id
                         and R.year = 2009);
```

"Find all courses that were offered at least twice in 2009"

```
select T.course id
from course as T
where not unique (select R.course id
                  from section as R
                  where T.course id= R.course id
                  and R.year = 2009);
```

Natural Join

The natural join operation operates on two relations and produces a relation as the result. Unlike the Cartesian product of two relations, which concatenates each tuple of the first relation with every tuple of the second, natural join considers only those pairs of tuples with the same value on those attributes that appear in the schemas of both relations

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column

Eg: List the names of instructors along with the course ID of the courses that they taught

select *name*, *course_id*
from *students, takes*
where *student.ID = takes.ID*;

Same query in SQL with "natural join" construct

select *name*, *course_id*
from *student* **natural join** *takes*

INNER Join

The INNER JOIN selects all rows from both participating tables as long as there is a match between the columns. An SQL INNER JOIN is same as JOIN clause, combining rows from two or more tables.

It is most common type of join. An SQL INNER JOIN return all rows from multiple tables where the join condition is met

The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Syntax:

SELECT column_name(s)

FROM table1 INNER JOIN table2

ON table1.column_name=table2.column_name;

Note: INNER JOIN is the same as JOIN.

CUSTOMERS table

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

ORDERS table

OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

Eg:

```
SELECT  ID, NAME, AMOUNT, DATE
  FROM CUSTOMERS
  INNER JOIN ORDERS
  ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

Result set will be :

ID	NAME	AMOUNT	DATE
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00

Outer Join

In inner join operation, some tuples in either or both of the relations being joined may be "lost".

The outer join operation works in a manner similar to the inner join operations we, but preserve those tuples that would be lost in a join, by creating tuples in the result containing null values.

The SQL OUTER JOIN returns all rows from both the participating tables which satisfy the join condition along with rows which do not satisfy the join condition.

- ✓ The **left outer join** preserves tuples only in the relation named before (to the left of) the **left outer join** operation
- ✓ The **right outer join** preserves tuples only in the relation named after (to the right of) the **right outer join** operation.
- ✓ The **full outer join** preserves tuples in both relations.

Left Outer Join

The LEFT JOIN keyword returns all rows from the left table (table1), with the matching rows in the right table (table2). The result is NULL in the right side when there is no match.

Left outer join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

The SQL LEFT OUTER JOIN returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in the right table; the join will still return a row in the result, but with NULL in each column from the right table.

SQL LEFT JOIN Syntax:

SELECT *column_name(s)*

FROM *table1* LEFT OUTER JOIN *table2*

ON *table1.column_name=table2.column_name*;

Eg:

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT OUTER JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20 00:00:00
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL

Right Outer Join

The RIGHT JOIN keyword returns all rows from the right table (table2), with the matching rows in the left table (table1). The result is NULL in the left side when there is no match.

The SQL RIGHT JOIN returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in the left table; the join will still return a row in the result, but with NULL in each column from the left table.

This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

SQL RIGHT JOIN Syntax:

SELECT column_name(s)

FROM table1 RIGHT OUTER JOIN table2

ON table1.column_name=table2.column_name;

Eg:

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

RIGHT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

ID	NAME	AMOUNT	DATE
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00

Full Outer Join

The FULL OUTER JOIN keyword returns all rows from the left table (table1) and from the right table (table2). The FULL OUTER JOIN keyword combines the result of both LEFT and RIGHT joins.

The SQL FULL JOIN combines the results of both left and right outer joins. The joined table will contain all records from both the tables and fill in NULLs for missing matches on either side

SQL FULL OUTER JOIN Syntax*:*

SELECT column_name(s)

FROM table1 FULL OUTER JOIN table2

ON table1.column_name=table2.column_name;

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

FULL OUTER JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20 00:00:00
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00

Note : If your Database does not support FULL JOIN (MySQL does not support FULL JOIN), then you can use UNION ALL clause to combine these two JOINS as shown below.

```
SELECT  ID, NAME, AMOUNT, DATE
  FROM CUSTOMERS
  LEFT JOIN ORDERS
  ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION ALL
  SELECT ID, NAME, AMOUNT, DATE
  FROM CUSTOMERS
  RIGHT JOIN ORDERS
  ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
```

Join Types and Conditions

Join types
inner join
left outer join
right outer join
full outer join

Join conditions
natural
on <predicate>
using $(A_1, A_2, \ldots, A_n)$

To distinguish normal joins from **outer joins**, normal joins are called **inner joins** in SQL. A join clause can thus specify inner join instead of outer join to specify that a normal join is to be used. The keyword **inner** is, however, optional. The default join type, when the join clause is used without the **outer** prefix is the **inner** join.

Thus,

*select **

from student join takes using (ID);

is equivalent to:

*select **

from student inner join takes using (ID);

Similarly, natural join is equivalent to natural inner join.

Any form of join (inner, left outer, right outer, or full outer) can be combined with any join condition (natural, using, or on).

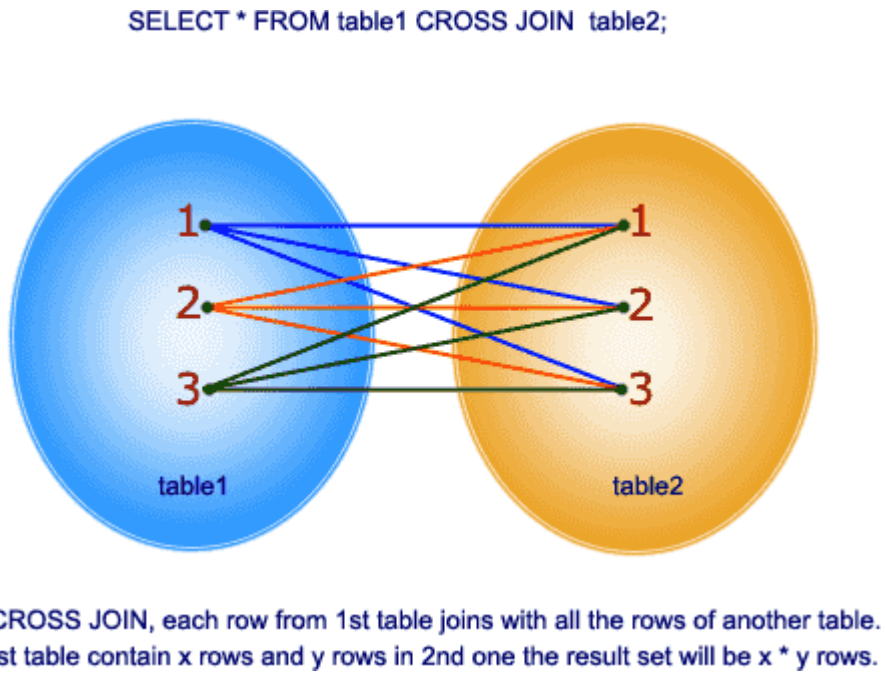
Cross Join

The SQL CROSS JOIN produces a result set which is the number of rows in the first table multiplied by the number of rows in the second table if no WHERE clause is used along with CROSS JOIN.This kind of result is called as Cartesian Product.

If WHERE clause is used with CROSS JOIN, it functions like an INNER JOIN.

Syntax:

```
SELECT *
FROM table1
CROSS JOIN table2;
```



In CROSS JOIN, each row from 1st table joins with all the rows of another table.
If 1st table contain x rows and y rows in 2nd one the result set will be x * y rows.

Example:

Here is an example of cross join in SQL between two tables.

Table: Foods

ITEM_ID	ITEM_NAME	ITEM_UNIT	COMPANY_ID
1	Chex Mix	Pcs	16
6	Cheez-It	Pcs	15
2	BN Biscuit	Pcs	15
3	Mighty Munch	Pcs	17
4	Pot Rice	Pcs	15
5	Jaffa Cakes	Pcs	18
7	Salt n Shake	Pcs	

Table: Company

COMPANY_ID	COMPANY_NAME	COMPANY_CITY
18	Order All	Boston
15	Jack Hill Ltd	London
16	Akas Foods	Delhi
17	Foodies.	London
19	sip-n-Bite.	New York

```
SELECT foods.item_name,foods.item_unit,
company.company_name,company.company_city
FROM foods
CROSS JOIN company;
```

Result:

ITEM_NAME	ITEM_UNIT	COMPANY_NAME	COMPANY_CITY
Chex Mix	Pcs	Order All	Boston
Cheez-It	Pcs	Order All	Boston
BN Biscuit	Pcs	Order All	Boston
Mighty Munch	Pcs	Order All	Boston
Pot Rice	Pcs	Order All	Boston
Jaffa Cakes	Pcs	Order All	Boston
Salt n Shake	Pcs	Order All	Boston
Chex Mix	Pcs	Jack Hill Ltd	London
Cheez-It	Pcs	Jack Hill Ltd	London
BN Biscuit	Pcs	Jack Hill Ltd	London
Mighty Munch	Pcs	Jack Hill Ltd	London
Pot Rice	Pcs	Jack Hill Ltd	London
Jaffa Cakes	Pcs	Jack Hill Ltd	London
Salt n Shake	Pcs	Jack Hill Ltd	London
Chex Mix	Pcs	Akas Foods	Delhi
Cheez-It	Pcs	Akas Foods	Delhi
BN Biscuit	Pcs	Akas Foods	Delhi
Mighty Munch	Pcs	Akas Foods	Delhi
Pot Rice	Pcs	Akas Foods	Delhi
Jaffa Cakes	Pcs	Akas Foods	Delhi
Salt n Shake	Pcs	Akas Foods	Delhi
Chex Mix	Pcs	Foodies.	London
.....			

Derived Relations: Views

In a relational database, all data are stored and accessed via relations. Relations that store data are called "base relations", and in implementations are called "tables". Other relations do not store data, but are computed by applying relational operations to other relations. These relations are sometimes called "derived relations". In implementations these are called "views". Derived relations are convenient in that though they may grab information from several relations, they act as a single relation. Also, derived relations can be used as an abstraction layer.

Views

SQL allows a “virtual relation” to be defined by a query, and the relation conceptually contains the result of the query. The virtual relation is not precomputed and stored, but instead is computed by executing the query whenever the virtual relation is used. Any such relation that is not part of the logical model, but is made visible to a user as a virtual relation, is called a **view**. It is possible to support a large number of views on top of any given set of actual relations.

A database view is a logical table. It does not physically store data like tables but represent data stored in underlying tables in different formats. A view does not require disk space and we can use view in most places where a table can be used. Since the views are derived from other tables thus when the data in its source tables are updated, the view reflects the updates as well. They also can be used by DBA to enforce database security.

- View is a virtual table, it does not contain data in it. It is map to the base table/s.
- In a table, we may require to prevent to all user from accessing all columns of table. So for the data security reasons views are created.
- One alternative solution is to create several table having appropriate no. of columns and assigned each user to each table.This provides well data security but it keeps redundant

data in tables. So it is not useful practically. So, views are created instead of it. It reduces redundant data.

- Views can be created from single table hiding some column/s r from multiple tables mapping all or some of the columns of the base tables. View is the simple and effective way of hiding columns of table for srecurity reasons.
- When view is referenced then only it holds data, so it reduces redundant data.
- When view is used to manipulate table, the underlying base table/s are completely invisible. This adds the level of data security.
- Since view can be created from multiple tables so it makes easy to query multiple tables because we can simply query view instead of multiple tables.
- View may be read only or updatable view. Read only view only allows to read data from view. Updatable view allow insert, update and delete operations on view.
- If view is created from multiple tables, it won't be updatable.
- If view is created without primary key and null columns then values/record cannot be inserted in view.

We define a view in SQL by using the create view command. To define a view, we must give the view a name and must state the query that computes the view. The form of the create view command is:

create view v as <query expression>;

where <query expression> is any legal query expression. The view name is represented by v.

Example: Following view contains the id, name, rating+5 and age of those Sailors whose age is greater than 30:

CREATE VIEW Sailor_view AS

SELECT sid, sname, rating+5, age

FROM Sailors

WHERE age>30;

Now by executing this query we get following view (logical table);

Select * from Sailor_view;

Sailor_view

Sid	sname	Rating+5	Age
1	Ajaya	17	33
2	Robin	16	43
9	Sanjaya	14	42
8	Rahul	11	76

Now any valid database operations can be performed in this view like in that of general table.

Example: creating view from single table.

```
CREATE VIEW   vw_emp   AS
     SELECT empno,  ename, job FROM emp;
  View coloumn can be renamed as below:
  CREATE VIEW vw_emp  AS
     SELECT empno "Employee no", ename " Employee name"
                                   Job "work" FROM  emp;
-    Creating view from multiple tables.
     CREATE VIEW vw_emp_info AS
          SELECT e.empno, e.ename, e.ejob, d.dname
                FROM emp e, dept d
          WHERE e.empno = d.dept no AND  e.sal>1000;
```

The attribute names of a view can be specified explicitly as follows:

create view *departments_total _salary*(*dept _name*, *total _salary*) **as**
select *dept _name*, **sum** (*salary*)
from *instructor*
group by *dept_ name*;

The preceding view gives for each department the sum of the salaries of all the instructors at that department. Since the expression sum(salary) does not have a name, the attribute name is specified explicitly in the view definition

Advantages of Views:

- Database security: view allows users to access only those sections of database that directly concerns them.
- View provides data independence.
- Easier querying
- Shielding from change
- Views provide group of users to access the data according to their criteria.
- Views allow the same data to be seen by different users in different ways at the same time.

Common Restrictions on View

- We cannot use delete statement on multiple table view.
- We cannot use insert statement unles all NOT NULL columns on underlying table are included.
- View must be created from single table to allow insert or update on view.
- If we use DISTINCT clause to create views, we cannot update or insert records within that view.

Common restrictions on updatable views

- For the views to be updatable, the view definantions must not include
  - Aggregate function
  - DISTINCT, GROUP BY or HAVING clause.
  - Sub-queries
  - Value expressions like bal*1.05
  - UNION, INTERSECT, OR, MINUS/EXCEPT clause.
- View can be destroyed by using DROP VIEW command

  DROP VIEW view_name;

Modification of the database

➔ Adding, removing and changing information in database.

- Insertion of new tuples into a given relation
- Deletion of tuples from a given relation.
- Updating of values in some tuples in a given relation

Deletion

- A delete request is expressed in much the same way as a query.
- We can delete only whole tuples; we cannot delete values on only particular attributes.
- Syntax:

delete from r

where P;

where P represents a predicate and r represents a relation.

- The delete statement first finds all tuples t in r for which P(t) is true, and then deletes them from r. The where clause can be omitted, in which case all tuples in **r** are deleted.
- A delete command operates on only one relation. If we want to delete tuples from several relations, we must use one delete command for each relation.

Examples

Example 1: suppose we need to remove all tuples of Sailors whose age is 32,

DELETE FROM Sailors

WHERE age=32;

Example 2: Suppose we need to delete all records from instructors table

delete from instructor

Example 3: Remove all tuples of Sailors whose age is less than 30 and rating greater than 7,

DELETE FROM Sailors

WHERE age<32 AND rating>7;

Updates

➔ If we need to change a particular value in a tuple without changing all values in the tuple, then for this purpose we use update operation.

➔ In certain situations, we may wish to change a value in a tuple without changing all values in the tuple. For this purpose, the update statement can be used. As we could for insert and delete, we can choose the tuples to be updated by using a query.

Syntax:

UPDATE table_name

SET <column i> = <expression i>;

Examples:

Example 1: Give a 5% salary raise to all instructors

update *instructor*
set *salary* = salary+ *salary* * 0.05

Example 2: Give a 5% salary raise to those instructors who earn less than 70000

update *instructor*
set *salary* = *salary+salary* * 0.05
where *salary* < 70000;

Example 3: Change address of employee to "Bhaktapur" whose employee id is "emp023".

update *employee*
set *address = 'Bhaktapur'*
where *emp_id='emp023'*;

The Rename Operation

- Generally, the names of the attributes in the result are derived from the names of the attributes in the relations in the **from** clause.
- We cannot, however, always derive names in this way, for several reasons:
  - First, two relations in the from clause may have attributes with the same name,in which case an attribute name is duplicated in the result.
  - Second, if we used an arithmetic expression in the select clause, the resultant attribute does not have a name.
  - Third, even if an attribute name can be derived from the base relations, we may want to change the attribute name in the result.
- Hence, SQL provides a way of renaming the attributes of a result relation. It uses the **as** clause, taking the form:

  *old-name **as** new-name*
- We can rename a table or a column temporarily by giving another name known as **Alias.**
- The use of table aliases is to rename a table in a specific SQL statement. The renaming is a temporary change and the actual table name does not change in the database. The column aliases are used to rename a table's columns for the purpose of a particular SQL query.
- Such alias names are identifiers which can be used as alternative names and also known as **tuple variables**, **correlation variables**, **correlation name** etc.

The basic syntax of a table alias is as follows.

```
SELECT column1, column2....
FROM table_name AS alias_name
WHERE [condition];
```

The basic syntax of a column alias is as follows.

```
SELECT column_name AS alias_name
FROM table_name
WHERE [condition];
```

Example

Consider the following two tables.

Table 1 – CUSTOMERS Table is as follows.

```
+----+----------+--------+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+--------+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+--------+-----------+----------+
```

Table 2 – ORDERS Table is as follows.

```
+-----+---------------------------+-------------------+------------+
|OID  | DATE                      | CUSTOMER_ID | AMOUNT |
+-----+---------------------------+-------------------+------------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
```

```
+----+----------------------------------+---------------------+-------------+
```

Now, the following code block shows the usage of a table alias.

```
SELECT C.ID, C.NAME, C.AGE, O.AMOUNT
  FROM CUSTOMERS AS C, ORDERS AS O
  WHERE  C.ID = O.CUSTOMER_ID;
```

This would produce the following result.

```
+---+---------------+-------+------------+
| ID | NAME     | AGE | AMOUNT |
+---+---------------+-------+------------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+---+---------------+-------+------------+
```

Following is the usage of a column alias.

```
SELECT ID AS CUSTOMER_ID, NAME AS CUSTOMER_NAME
  FROM CUSTOMERS
  WHERE SALARY IS NOT NULL;
```

This would produce the following result.

```
+--------------------+-----------------------+
| CUSTOMER_ID  | CUSTOMER_NAME |
+--------------------+-----------------------+
|            1 | Ramesh        |
|            2 | Khilan        |
|            3 | kaushik       |
|            4 | Chaitali      |
|            5 | Hardik        |
|            6 | Komal         |
|            7 | Muffy         |
+--------------------+-----------------------+
```

NOTE : Keyword **as** is optional and may be omitted.

CUSTOMERS AS C ≡ CUSTOMERS C

Tuple Variables: C and O in following query

```
SELECT C.ID, C.NAME, C.AGE, O.AMOUNT
  FROM CUSTOMERS AS C, ORDERS AS O
  WHERE  C.ID = O.CUSTOMER_ID;
```