**DBMS (R20) 2nd Year 1st Semester 2021-22 AY**

**Unit-3 Syllabus**

**Translating E-R diagrams to Relations**: Translating Entity Sets, Relationships, Weak Entity Sets, Class Hierarchies, and Aggregation.

**SQL**: Creating Tables with Relationship, Implementation of Key and Other Integrity Constraints, Nested Queries, Sub Queries, Grouping, Aggregation, Ordering, Implementation of Different Types of Joins, Views (Updatable and Non-Updatable), Relational Set Operations.

**Translating E-R diagrams to Relations**:

i) ***Translating Entity sets to Relations***: Consider the following entity set. The name of entity se becomes name of the relation. Each attribute forms a column of the relation.

Students

**Entity Set**

Create table Students (Std\_ID varchar(10) primary key, Std\_Name varchar(15), Dr\_No varchar(15), St\_Name varchar(15), Town varchar(15));

Create Table Std\_Mobiles(Std\_ID varchar(10) references Students on delete cascade, Mobile\_Number number(10) Unique, Primary key(Std\_ID,Mobile\_Number));

ii) ***Translating Relationships without any******constraints***:

Employees

Reports\_To

Supervisor Subordinate

**Unary Relationship**

Create table Reports\_To(Supervisor\_SSN number(4) references Employees, Subordinate\_SSN number(4) references Employees on delete cascade, primary key (Supervisor\_SSN, Subordinate\_SSN));

Employees

Departments

Works\_in1

Binary Relationship

First, create tables for Employees and Departments entity sets. Then create table for relationship as follows:

Create table Works\_in1(Emp\_ID varchar(10) references Employees on delete cascade, Dept\_Name varchar(15) references Departments,primary key(Emp\_ID,Dept\_Name));

iii) ***Translating Relationships with key constraints***:

Works\_in

M 1

Departments

Employees

Manages

1 1

Two Relationships between two Entity sets

First, create Departments table. Next create table Employees as follows:

Create table Employees( Emp\_ID varcar(10) primary key, E\_Name varchar(15), lot number(2), Dept\_Name varchar(15) not null, foreign key(Dept\_Name) references Departments);

iii) ***Translating Relationships with participation constraints***:

Works\_in

M 1

Departments

Employees

1 1 1

Manages

Representing Cardinalities and Participation constraints

Employees entity set has total participation in the “Works\_in” relationship. We can include the working department of employee in Employees table and make the column not null.

Create table Employees (Emp\_ID varchar(5) primary key, E\_Name varchar(15), lot number(2), Dept\_Name varchar(15) not null, foreign key(Dept\_Name) references Departments);

Total participation of Departments cannot be expressed using ICs alone. Advanced ICs called Assertions are used for that purpose.

iv) **Translating weak entity sets**:

Loan\_Payment

1 M

Payments

Loans

A Weak Entity Set

First, create table for strong entity set. Next, create table for weak entity set as follows:

Create table Payments (Loan\_Number varchar(15) references loans on delete cascade, Payment\_number number(3), P\_Amount number(5), Day Date, primary key(Loan\_Number, Payment\_Nummber));

No table is required for Identifying relationship.

v) ***Translating Hierarchies***:

Contract\_Emps

Hourly\_Emps

ISA

Employees

Representing Class Hierarchies

To translate hierarchies into tables, one of the two approaches can be followed. In one way, create three tables. First table for Employees entity set with all its attributes. Create second table for Hourly Employees with primary key of Employees table and additional properties of Hourly employees. Create third table for contract employees.

Create table Employees (Emp\_ID varchar(5) primary key, e\_name varchar(15), lot number(2));

Create table Hrly\_Emps (Emp\_ID varchar(5) primary key, Hrly\_wage number(4), No\_of\_Hrs number(5), foreign key(Emp\_ID) references Employees on delete cascade);

Similarly create table for “Contract\_Emps” also.

In second way, create only two tables for sub class entity sets. Avoid creation of Employees table. Include Employee table properties also in sub class tables.

vi) ***Translating Aggregation into tables***:

Create tables for three entity sets: *Employees*, *Departments* and *Projects*. Now, create a table for binary relationship *Sponsors* along with its descriptive attribute.

Create table Sponsors (Dept\_Name varchar(20) references Departments, Proj\_Name varchar(20) references Projects, Since date, primary key(Dept\_Name, Proj\_Name));

Now create table for *Monitors* as follows:

Create table *Monitors* (Dept\_Name varchar(20), Proj\_Name varchar(20), Emp\_ID references Employees, primary key(Dept\_Name, Proj\_Name, Emp\_ID), foreign key (Dept\_Name, Proj\_Name) references Sponsors on delete cascade);

Sponsors

Projects

Departments

**Representing Aggregation**

Employees

Monitors

**Creating Tables with Integrity Constraints**:- Consider the following schema to illustrate the application of various column level constraints.

**Students**(Std\_ID Char(6) Primary key, Std\_Name Varchar(15) Not Null, Email\_ID Varchar2(20) Unique and Not Null, Gender Varchar2(6)) Restrict Gender column to take either “Male” or “Female”.

Table for the above schema is created as follows.

Create Table Sudents( Std\_ID Char(6) Primary key, Std\_Name Varchar(15) Not Null, Email\_ID Varchar2(20) Unique Not Null, Gender Varchar2(6), Check(Gender In(‘Male’,’Female’)));

***Set Manipulation Constructs***:

These are used to combine the results of two queries. The results of two queries must be compatible i.e., both the results should have equal number of columns and the types of columns taken in order must be same. The sizes and names of corresponding columns may be different.

Let Q1 and Q2 are two SQL queries and R1 and R2 are their results. Let R1 and R2 are compatible.

-> Q1 ***union*** Q2 returns those records that are either in R1 or in R2.

-> Q1 ***intersect*** Q2 returns those records that are in both R1 and R2.

-> Q1 ***except*** Q2 returns those records that are in R1 but not in R2.

The key word “***except***” is not supported by some Oracle versions. Instead of that, “***minus***” is used.

15) Find the IDs of sailors who have reserved both a Red and a Green boat.

Select R.Sid from Reserves R, Boats B where R.Bid = B.Bid and B.colour = ‘Red’

***intersect***

Select R1.Sid from Reserves R1, Boats B1 where R1.Bid = B1.Bid and B1.color = ‘Green’;

16) Find the IDs of sailors who have reserved a Red boat but not a Green boat.

Select R.Sid from Reserves R, Boats B where R.Bid = B.Bid and B.colour = ‘Red’

***minus***

Select R1.Sid from Reserves R1, Boats B1 where R1.Bid = B1.Bid and B1.colour = ‘Green’;

17) Find the IDs of sailors who have reserved boat 103 or who have a rating above 7.

Select R.Sid from Reserves R where R.Bid = 103

***union***

Select S.Sid from Sailors S where rating > 7;

***Set Comparison Operators*** (**Any** and **All**):

These operators are used to compare a data value against each element of a list of data values of same type as that of comparing value.

Any relational operator in the set of { =, !=, >, <, >=, <=} is used for comparison.

**Syntax**:

col\_name *operator* ***set\_operator*** (sub query). Before “*col\_name*”, the key word ***where*** should occur.

**Ex**:

Where rating > **Any** (Select rating from Sailors)

---“Any” returns true if the comparison is true on at least one element in the list.

---“All” returns true if the comparison is true against every element in list.

18) Find the highest rating among all sailors.

Select Rating from Sailors where Rating >= **All** (Select Distinct S.Rating from Sailors S);

19) Find IDs and names of sailors whose rating is better than some sailor named ‘Smith’.

Select Sid,Sname from Sailors where rating >= **Any** (Select S.Rating from Sailors S where S.Sname = ‘Smith’);

20) Find the details of oldest sailors among the sailors of rating level 8.

Select \* from Sailors where Age >= All (Select S.Age from Sailors S where S.Rating = 8)) and Rating = 8;

***Nested Queries***:

A nested query is a query which has a second query embedded in it. The embedded query is called ***sub query***and the outer query is called ***main query***. The query in total is called ***nested query***.

A sub query itself may be a nested query resulting in a deeply nested query. Embedding of queries can be done to any depth. The evaluation process is as follows:

**Step 1**: If there are multiple tables in outer query, their cross product is computed.

**Step 2**: The first row of the table in the outer query is considered and the sub query is executed for that row. Then the second row is taken and the sub query is executed again. This process continues on all rows of the outer query table.

The sub query result generated in each pass, qualifies the current row of the outer table. If the row is qualified, the values of the columns mentioned in the select list of main query are displayed for that row.

The sub query result is treated as a table generated using a SQL query. Hence, sub queries may be typically used in the “*from*” clause of a SQL query in addition to base tables. The sub query result is the same for every row of outer query table in some nested queries. Even then, the sub query result is executed once for each pass. It’s a common way of execution.

--”***In***” and “***Exists***” are two operators used with nested queries.

“***In”*** is used to check for the presence of an element in a list of elements of same type. The result returns ***true*** if the element is ***present***. Else it returns ***false***. “***Not In***” is used in the opposite sense.

21) Find the IDs and names of sailors who have reserved boat with ID 103

Select Sid,Sname from Sailors where Sid ***In*** ( Select Sid from Reserves where Bid=103);

Each row of Sailors table is considered one at a time. The sub query returns a list of sailors who have reserved boat 103. If the *Sid* of current row is present in the list returned by sub query, ***In*** returns True for that row. Hence, his ID and name are displayed in the final result.

Observe that when we search for the presence of a *Sid*, the sub query should return a list of *Sid* values only. Also, for queries using ***In*** as the operator, the sub query always returns the same list.

22) Find the IDs of sailors who have not reserved any Red boats.

Select Sid from Sailors

Minus

Select Sid from Reserves where Bid In (Select Bid from Boats where Colour = ‘Red’);

The second query uses a sub query to return a list of Sid values that have reserved Red boats. Then the list is subtracted from the list of all sailors to generate the required result.

23) Find the IDs of sailors who have reserved only a Red boat.

Select R.Sid from Reserves R where R.Sid ***Not In*** (Select Sid

from Reserves

where Bid ***In*** (Select Bid from

Boats where

Colour!= ‘Red’);

\*\*\* Observe that by replacing **Not In** with **In** and the sub query condition **Colour!=’Red’** with **Colour=’Red’**, we cannot get the correct result.

***Correlated Queries***:

In some Queries, the sub query result depends on the current row of the outer query table. In such queries, attributes of outer query row appears in sub query. Such an occurrence is called ***correlation*** and hence the queries are called ***correlated queries***.

The operator “***Exists***” is used with correlated queries. ***Exists*** checks whether the result of sub query is empty or not. It does not consider the instance of the result. Hence, the select clause of sub query can contain any column, usually a “\*”. *Exists* returns ***True*** if the sub query result is ***non-empty***. Otherwise, it returns ***False***. “***Not Exists***” works in the opposite way.

\*\*\* When we use ***In***, the syntax will be: “…Where ***Col\_Name*** In (Sub Query)”.

When we use Exists, the syntax will be “...Where Exists (Sub Query)”.

Observe that there is **no column** **name** between *Where* and *Exists*.

24) Find IDs and names of sailors who have reserved boat 104.

Select S.Sid,S.Sname

from Sailors **S**

where ***Exists*** (Select \* from Reserves R where R.Bid = 104 and R.Sid = **S**.Sid);

During execution, first row of Sailors table is considered at first. With that *Sid* value, the sub query is evaluated. If this sailor has reserved boat 104, then the sub query result is non empty. Hence, this sailor is displayed in the final result. This process continues for every row. Depending up on the current row of outer query table, the sub query result may be empty or non empty. Only those rows for which the sub query result is non empty are considered for final result.

The occurrence of **S** in both outer and sub queries is called ***correlation***.

25) Find the IDs and names of sailors who have reserved all boats.

Select S.Sid,S.Sname

from Sailors **S**

where ***Not Exists*** (Select Bid from Boats Minus Select R.Bid from

Reserves R where R.Sid = **S**.Sid)

***Aggregate Functions***:

These functions are used to aggregate or consolidate the data present in list.(usually numbers). There are five functions supported. Consider a list L={5, 10, 5, 3, 3, 6}.

---Sum of elements in the list L is: 32. Sum of unique values in the list is: 24

---Average of elements in the list is: 5.33. Average of unique values in the list is: 6

---Count of elements in the list is: 6. Count of unique values in the list is: 4

---Maximum value in the list is: 10

---Minimum value in the list is: 3

These five ways of summarizing data are called aggregate functions.

---***Sum***(L) – returns the sum of all elements in the list L.(Applied usually on numbers)

--- ***Sum***(*Distinct* L)—returns the sum after removing duplicates.

--- ***Avg***(L) – returns the average value of all elements in list L.(Applied usually on numbers)

--- ***Avg***(*Distinct* L)—returns the average value of distinct elements after removing duplicates.

---***Count***(L) – returns count of all elements in the list L.(Applied on any type of elements)

---***Count***(*Distinct* L)—returns the count of distinct elements.

---***Max***(L) – returns the maximum value among list L.(Applied on numbers, strings and dates)

---***Min***(L) – returns the minimum value among list L.(Applied on numbers, strings, dates)

\*\*\****Max***() and ***Min***() functions returns the same value with or with out distinct.

Ex:

L = {2,4,5,5,6,4}

Sum(L) = 26 Sum(Distinct L) = 17

Avg(L) = 4.33 Avg(Distinct L) = 4.25

Count(L) = 6 Count(Distinct L) = 4

Max(L) = 6 Min(L) = 2

Meaningful functions:

Rating – Avg, Max, Min, Count

Age – Avg, Max, Min, Count

Salary – Avg, Sum, Max, Min, Count

Dob – Count, Min (Oldest), MAX (Youngest)

26) Find the count of sailors.

Select Count(\*) from Sailors;

27) Find Minimum, Maximum, Average ratings of sailors table.

Select Min(Rating), Max(Rating), Avg(Rating) from Sailors;

28) Find the IDs of Sailors with highest rating.

Select Sid

from Sailors

where Rating = (Select Max(rating) from Sailors);

29) Find the Ids and Rating for sailors with best rating.

Select S.Sid,S.Rating

from Sailors S

where S.Rating = (Select Max(Rating) from Sailors);

“***Group by***” and “***Having***” Clauses:

Consider the following query:

Find the highest age for each rating level. The answer for this query is:

Select max(age) from sailors where rating=1,2,3…

In the above query, we need to provide one rating level at a time and we will get the highest age for that rating. The drawback in the above solution is: First, we need to know in advance, how many rating levels exist in the table and what those ratings are. Also, we get the maximum age for only one rating at a time. A solution to this problem is the use of “group by” clause.

Syntax:

select select\_list (Step 4)

from from\_list (Step 1)

where qualification (Step 2)

group by grouping\_list (Step 3)

having group qualification; (Step 5)

In Step 1, cross product of the tables is computed. In Step 2, unwanted records are eliminated. In Step 3, qualified records are divided into groups based on grouping criterion. In Step 4, aggregation is applied on specified columns to generate one value per group. In Step 5, unwanted groups are eliminated from final result by applying qualification specified in having clause.

**Note**:

i) Every column that appears in the Select list (other than aggregate columns) must appear in the grouping list.

ii) Columns that appear in the grouping list need not appear in the Select list.

iii) In the absence of ‘group by’ clause, the entire table is considered as a single group. In this case the Select list contains only aggregate columns.

iv) The ‘HAVING’ clause qualification used to eliminate unwanted groups must be single valued per group.

30) For each rating level display the highest age.

Select Rating, MAX(Age)

from Sailors group by Rating;

31) For each sailor, find the count of reservations made.

Select Sid, Count(\*)

from Sailors group by Rating;

32) For each sailor display the count of different boats reserved by him.

Select Sid,Count(Distinct Bid)

from Reserves group by Sid;

33) For each Red boat, find the count of reservations made.

Select R.Bid, Count(\*)

from Boats B,Reserves R

where B.Bid=R.Bid group by R.Bid having B.Colour=’Red’;

The above way is not allowed because the colour of a boat is determined from *Bid* which cannot be accepted as group qualification. The correct way of answering is as follows:

Select R.Bid, Count(\*)

from Boats B,Reserves R

where B.Bid=R.Bid and B.Colour=’Red’ group by R.Bid;

34) Find the average age of sailors who are above 18 years for each rating level that has at least 2 *such* sailors.

Select Rating , Avg(Age)

from Sailors

where Age>18 group by Rating HAVING count(\*)>=2;

35) Find the average age of sailors who are above 18 years for each rating level that has at least 2 sailors.

Select Avg (S.Age), S.Rating

from Sailors S

where S.Age > 18 group by S.Rating

having (Select Count(\*) from Sailors S1 where S1.Rating = S.Rating) >=2;

36) Find the rating levels for which the average age is the minimum among all ratings.

Select Temp.Rating, Temp.Avgage

from (Select S.Rating, Avg(S.Age) as AvgAge from Sailors S

group by S.rating) as Temp

where Temp.AvgAge = (Select Min (Temp.AvgAge) from Temp);

37) Find the rating levels in which all sailors are above 25 years.

Select S.Rating from Sailors S where S.Age > 25

group by S.Rating having Count(\*) = ( Select COUNT(\*) from Sailors S1 where S1.rating = S.rating);

***Sorting Results***: SQL supports displaying results in a sorted order based on a column. When the values in a column are sorted, the entire record associated with a value will move to the sorted position. i.e., tuples won’t be split due to sorting. Both ascending and descending sorting is allowed. When the result of a SQL query is sorted based on column ***C1***, and if C1 has some values repeated, then the result can be sorted again based on a second column ***C2*** retaining first sort order.

38) Display all sailors in the ascending order of their rating.

Select \* from sailors order by Rating asc;

39) Display all sailors in the descending order of their rating and then in the ascending order of age within each rating.

Select \* from sailors order by Rating desc, Age asc;

***Sorting Months and Days in Calendar order***: Columns of type “Date” can be sorted in the calendar order. The following examples illustrate sorting months and week-days in calendar order.

Select DoB,To\_Char(DoB,'Month') as Birth\_Month from Familymembers order by to\_char(DOB,'MM');

DOB BIRTH\_MON

--------- ---------

27-JAN-88 January

24-MAR-87 March

26-APR-91 April

16-MAY-87 May

20-JUN-87 June

22-JUL-90 July

01-AUG-93 August

17-OCT-90 October

8 rows selected.

\*\*\* Observe that DoB is not sorted as per year. Only months are sorted.

Select To\_Char(DoB,'fmDay') from Familymembers Order By To\_Char(DoB,'D');

TO\_CHAR(D

---------

Sunday

Sunday

Tuesday

Wednesday

Wednesday

Friday

Saturday

Saturday

8 rows selected.

\*\*\* Week-Day of DoB is retrieved for each row and all days are sorted in calendar order.

***Custom Sorting of Strings***: When strings are to be sorted in custom order other than alphabetical order, we use ***case*** to specify the position of each string in the sorted list. In the following example, *Hire\_Month* is the column name. In this column, months are considered as just strings and not as calendar months.

Select Empno,Hire\_Month from Emp Order By Case

When Hire\_Month='January' Then 1

When Hire\_Month='February' Then 2

When Hire\_Month='March' Then 3

When Hire\_Month='April' Then 4

When Hire\_Month='May' Then 5

When Hire\_Month='June' Then 6

When Hire\_Month='July' Then 7

When Hire\_Month='August' Then 8

When Hire\_Month='September' Then 9

When Hire\_Month='October' Then 10

When Hire\_Month='November' Then 11

When Hire\_Month='December' Then 12

End;

***Null values***: In some cases column values for some records may be not available. For example, when a sailor has newly joined the club, he may not be given any rating. As a second example, the maiden name for men is not applicable. SQL provides a special value called “***null***” to use in such cases. A null value represent that the column value is either *unknown* or *not applicable*. When there are definite values, any comparison returns either true or false. But, in the presence of null values, three valued logic with values {True, False, Unknown} is used. Comparing a rating 8>null returns unknown. Any comparison involving at least one null value returns unknown as its result. To check whether a column value is null, the operator “is null” is used in the where clause. “…..where rating is null”; Using *unknown*, Logical connectives AND, OR and NOT are redefined as follows:

True AND unknown---Unknown, Unknown AND Unknown---Unknown, False AND Unknown---False

True OR unknown---True, Unknown OR Unknown---Unknown, False OR Unknown---Unknown

NOT Unknown---Unknown

Rows having null values are disqualified by the where clause condition. For example, consider a sailor has rating value null. The where clause condition “…where rating=8” eliminates this record. The aggregate function count(\*) counts null values also. All other aggregate functions discard null values.

***Views***: A view is a virtual relation whose rows are not explicitly stored in the database. They are computed from the view definition as and when needed. When a view is created, its definition is stored in the database. When the view is accessed through an SQL query, its instance is computed from the instances of underlying base tables. Since a view is also a relation, it can be used in the “*from”* clause of an SQL query along with base tables. If the view instance is stored at the time of its creation, it may get outdated later when the instances of underlying base tables are changed. Hence, view instances are not stored.

However, some versions of SQL stores view instances with a promise to update stored view instance whenever the instances of underlying base tables are changed. Such views are called ***materialized views***.

When a view is created, DML operations can be performed on the underlying base table through the view. Such views are called ***updateable views***. For a view to be updateable, following conditions should be met:

--It should contain only one base table.

--No aggregate functions should present in the select clause of the view.

--All columns of the base table that are *not null* should be included in the view.

***Advantages of views***: Views provide the following advantages:

1. They provide security to the data by allowing user access to a subset of database only.
2. They simplify query writing.
3. They provide a simple view of a complex database.

Ex: Create a view that contains only Sailors IDs and age.

Create view S\_ID\_Age as select Sid,Age from sailors;

insert into S\_ID\_Age values(102,33); This statement inserts a sailor record into Sailors table with null values for rating and name columns.

Select \* from S\_ID\_age; ( To display instance of view).

A view can be created from multiple tables also. Following statement creates a view that contains Sid, Sname and Bid for each reservation made.

Create view S\_ID\_Name\_Bid as select S.sid,S.Sname,R.Bid from Sailors S, Reserves R

Where S.Sid=R.Sid;

***Joins***: A join provides a simple way to write queries. A join operation is equal to cross product of tables followed by selection and projection. For example, consider the query “Find the IDs and names of Sailors who reserved boat 103”. Following statements show writing query with and without join.

Select S.Sid,s.Sname from Sailors S, Reserves R where S.Sid=R.Sid and R.bid=103; (Without Join)

Select Sid,Sname from Sailors ***natural inner join*** Reserves where Bid=103; (with Join)

In the above statement, “Sailors ***natural inner join*** Reserves” is equal to performing cross product of Sailors and Reserves tables and then applying where clause condition “Sailors.Sid=Reserves.Sid” followed by removing either S.Sid or R.Sid in the select clause.

To use joins, the two tables should have a common attribute. i.e., attribute with same name in both the tables. Otherwise, Join simply performs their cross product and returns the result.

***Types of joins***: Various joins are:

---*Natural Inner Join*: T1 natural inner join T2 considers only those rows from T1 for which a matching row is there in T2 on common attribute. If more rows of T2 match with one row of T1, then all those rows are considered. The common attribute appears only once in the result.

---*Natural Left Outer Join*: T1 natural left outer join T2 considers all rows from T1. If one or more matching rows are found in T2, then they are considered in the final result. If some records of T1 are left unmatched, they are considered once in the result by placing *null* values in the columns specific to T2.

---*Natural Right Outer Join*: T1 natural right outer join T2 considers all rows from T2. If one or more matching rows are found in T1, then they are considered in the final result. If some records of T2 are left unmatched, they are considered once in the result by placing *null* values in the columns specific to T1.

---*Natural Full Outer Join*: T1 natural full outer join T2 considers all rows from both the tables T1 and T2. Those rows that have matching are displayed as usual. Those rows from T1 that does not match any row of T2 are considered once by placing null values in columns specific to T2. Similarly, those rows from T2 that does not match any row of T1 are considered once by placing null values in columns specific to T1.

Following SQL examples illustrate joins. Here, X is the common attribute.

SQL> select \* from join\_ex1;

X Y

------ -----

1000 Rose

1001 Ramu

1002 Sagar

1003 Sagar

1004 Lily

SQL> select \* from join\_ex2;

Z X

---- -----

abcd 1000

abcd 1001

pqrs 1001

pqrs 1002

xyzw 1005

SQL> select \* from join\_ex1 natural inner join join\_ex2;

X Y Z

----- ----- ----

1000 Rose abcd

1001 Ramu abcd

1001 Ramu pqrs

1002 Sagar pqrs

SQL> select \* from join\_ex1 natural left outer join join\_ex2;

X Y Z

------ ----- ----

1000 Rose abcd

1001 Ramu abcd

1001 Ramu pqrs

1002 Sagar pqrs

1003 Sagar

1004 Lily

6 rows selected.

SQL> select \* from join\_ex1 natural right outer join join\_ex2;

X Y Z

----- ----- ----

1000 Rose abcd

1001 Ramu pqrs

1001 Ramu abcd

1002 Sagar pqrs

1005 xyzw

SQL> select \* from join\_ex1 natural full outer join join\_ex2;

X Y Z

---- ----- ----

1000 Rose abcd

1001 Ramu abcd

1001 Ramu pqrs

1002 Sagar pqrs

1003 Sagar

1004 Lily

1005 xyzw

**Note**: Blanks shows ***null*** values.

---When two tables do not have a common attribute, natural inner join simply returns cross-product.

Select \* from Join1; Select \* from Join2;

SID SNAME CID SNO

------ ---------- --- ----------

21 Ramesh C01 21

22 Krishna C01 22

23 Madhulika C02 22

Select \* from Join1 Natural Inner Join Join2;

SID SNAME CID SNO

------- ---------- --- ----------

21 Ramesh C01 21

21 Ramesh C01 22

21 Ramesh C02 22

22 Krishna C01 21

22 Krishna C01 22

22 Krishna C02 22

23 Madhulika C01 21

23 Madhulika C01 22

23 Madhulika C02 22

Select \* from Join1 Inner Join Join2 On SID=SNO; (***Natural*** Key word not there)

SID SNAME CID SNO

---------- ---------- --- ----------

21 Ramesh C01 21

22 Krishna C01 22

22 Krishna C02 22

------XXXXX------