|  |
| --- |
| **TASK - 01** |

Create following tables (schema of University enterprise) with primary key and foreign key constraints.Bold indicates primary keys. Foreign keys are to be understood (See data).

classroom(**building, room\_number**, capacity)

department(**dept\_name**,building, budget)

course(**course\_id**, title, dept\_name,credits)

instructor(**ID**,name,dept\_name,salary)

section(**course\_id,sec\_id,semester**,year,building, room\_number,time\_slot\_id)

teaches(**ID, course\_id,sec\_id,semester,year**)

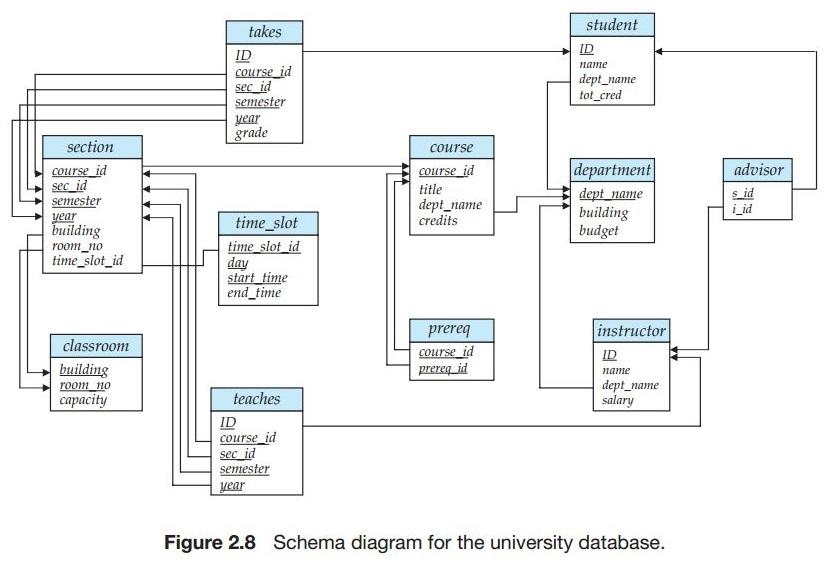
student(**ID**,name,dept\_name,tot\_cred)

takes(**ID,course\_id, sec\_id,semester, year**,grade)

advisor(**s\_ID**,i\_ID)

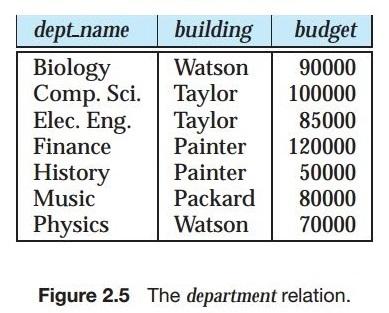
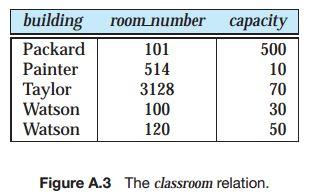
timeslot(**time\_slot\_id,day,start\_time**,end\_time)

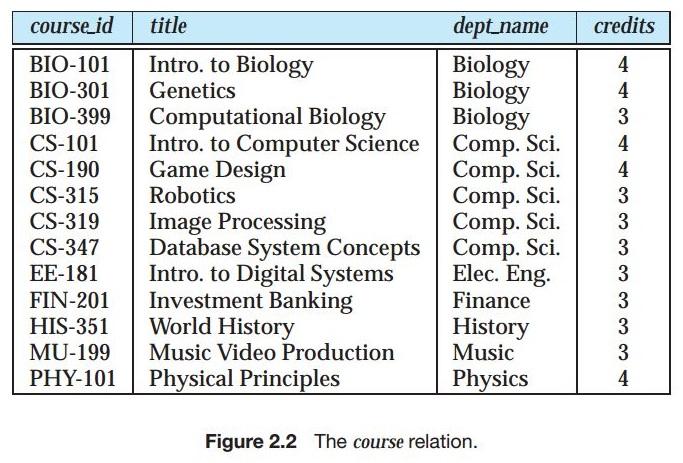
prereq(**course\_id,prereq\_id**)

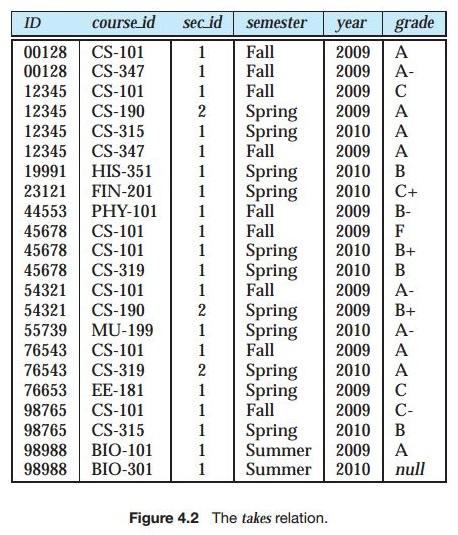
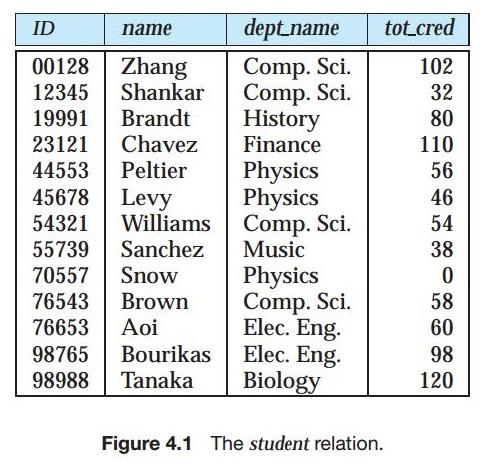
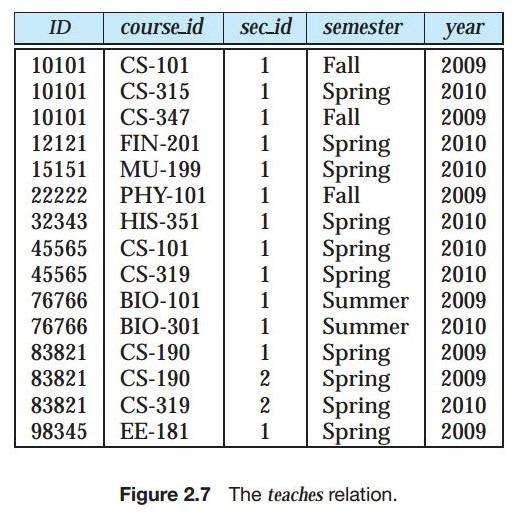
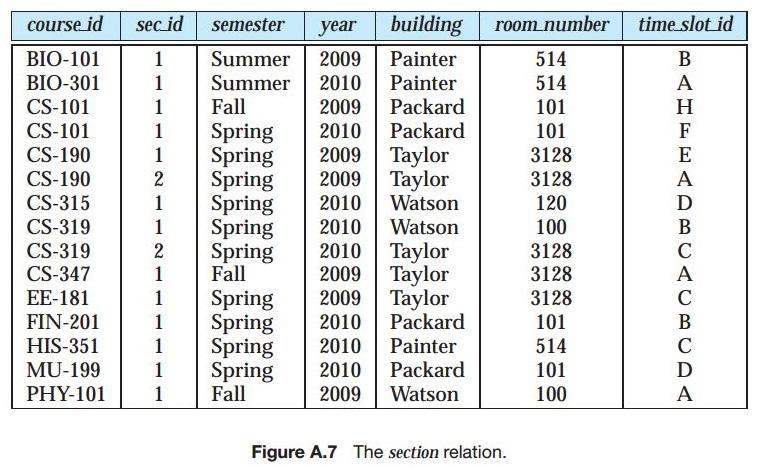
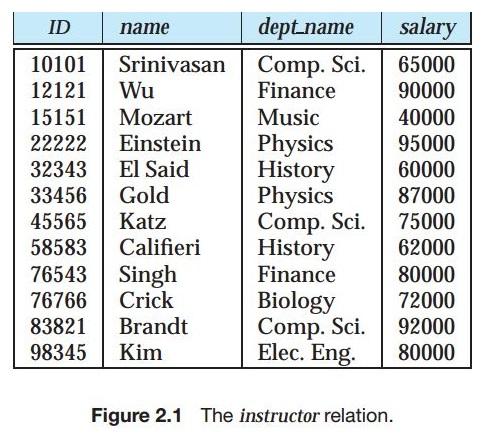


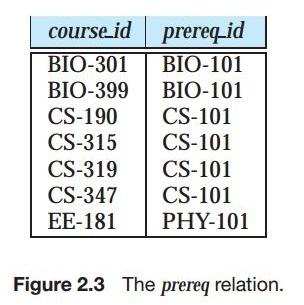
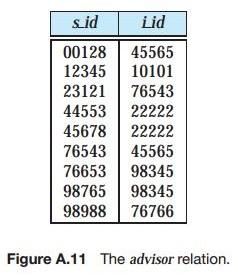
|  |
| --- |
| **TASK - 02** |

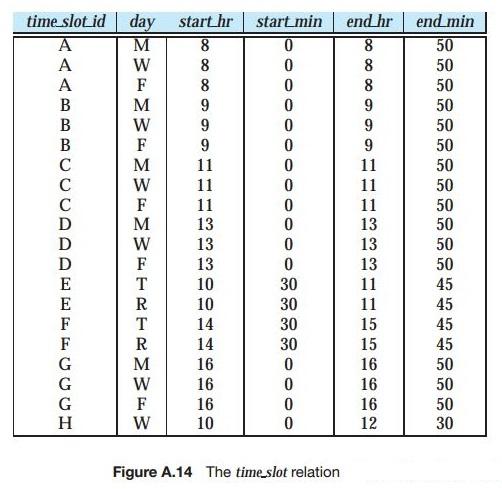
Then Insert following data shown in the tables:











|  |
| --- |
| **TASK - 03** |

Show and run the above scripts and then perform the following queries:

1. Find the names of all departments with instructor, and remove duplicates.
2. Find the names of all instructors in the History department.
3. Find the instructor\_ID and department\_name of all instructors associated with a department with budget of greater than $95,000.
4. Find all instructors in Computer Science department with salary more than$80,000.
5. List the names of instructors along with the titles of courses that they teach.
6. Find the names of all instructors who have a higher salary than some instructor in ‘Computer Science’.
7. Find the names of all instructors whose salary is greater than at least one instructor in the Biology department.
8. Find the names of all departments whose building name includes the substring ‘Watson’.
9. List in alphabetic order the names of all instructors.
10. Find the names of instructors with salary amounts between $90,000 and $100,000.
11. Find the set of all courses taught either in Fall 2009 or in Spring 2010Semester, or both.
12. Find the set of all courses taught in the Fall 2009 as well as in Spring 2010 Semester.
13. Find all courses taught in the Fall 2009 semester but not in the Spring 2010Semester.
14. Find all instructors whose salary is null.
15. Find courses offered in Fall 2009 and in Spring 2010Semester.
16. Find courses offered in Fall 2009 but not in Spring 2010Semester.
17. Find the total number of (distinct) students who have taken course sections taught by the instructor with ID 10101.
18. Find the names of instructors with salary greater than that of some (at least one) instructor in the Biology department.
19. Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.
20. Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester.
21. Find all students who have taken all courses offered in the Biology department.
22. Find all courses that were offered at most once in 2009

**Compiled Reference**

|  |
| --- |
| **SELECT … FROM … WHERE …** |

The SELECT statement allows you to retrieve records from one or more tables in your database.

The syntax for the SELECT statement is:

SELECT columns  
FROM tables  
WHERE predicates;

In general, the meaning of a SQL query can be understood as follows:

1. Generate a Cartesian product of the relations (if there) listed in the from clause
2. Apply the predicates specified in the where clause on the result of Step 1.
3. For each tuple in the result of Step 2, output the attributes (or results of expressions) specified in the select clause.

The more-or-less full syntax of the SELECT statement is:

SELECT column-list

FROM table-join-expression

[WHERE expression]

[GROUP BY column-list]

[HAVING aggregate-Boolean-expression]

[ORDER BY column-list];

The meaning of each clause is as follows:

* The SELECT clause defines which columns will appear in the result set.
* The FROM clause specifies which tables will provide the data for the result set, and also how they should be joined together.
* The WHERE clause determines which rows will appear in the result set. The absence of the WHERE clause means that all rows will appear.
* The GROUP BY clause creates group on basis of related columns.
* The HAVING clause is only necessary when the GROUP BY clause is in place, and is used to filter out rows from the grouped result set.
* The ORDER BY clause sorts the result set.

**YOU MUST KNOW:**

* SQL statements are not case sensitive. E.g. Name ≡ NAME ≡ name
* SQL statements can be on one or more lines.
* Keywords cannot be abbreviated or split across lines.
* Clauses are usually placed on separate lines.
* Indents are used to enhance readability.

**MORE TERMS:**

* KEYWORD:
* CLAUSE:
* STATEMENT:

**Combining the "AND" and "OR" Conditions in WHERE clause**

The AND and OR conditions can be combined in a single SQL statement. When combining these conditions, it is important to use brackets so that the database knows what order to evaluate each condition. See the examples:

SELECT \*FROM suppliers  
WHERE (city = 'New York' and name = 'IBM')or (city = 'Newark');

This would return all suppliers that reside in New York whose name is IBM and all suppliers that reside in Newark. The brackets determine what order the AND and OR conditions are evaluated in. Test the following SQL:

SELECT \*FROM suppliers  
WHERE city = 'New York' and name = 'IBM' or city = 'Newark';

**Eliminate Duplicates with DISTINCT**

If a result set produced by a SELECT statement contains duplicate rows, these can be eliminated by using the DISTINCT keyword in the query immediately following the SELECT keyword. For example,

SELECT DISTINCT model

FROM aircraft

Note that DISTINCT applies to entire rows in the result set and not individual columns. Indiscriminate use of DISTINCT can occasionally produce misleading results, particularly in those cases when it is perfectly allowable for duplicate rows to exist in a result set.

**Specify range using BETWEEN...AND**

The following is an SQL statement that uses the BETWEEN function:

SELECT \*FROM suppliers  
WHERE supplier\_id between 5000 AND 5010;

This would return all rows where the supplier\_id is between 5000 and 5010, inclusive. It is equivalent to the following SQL statement:

SELECT \*FROM suppliers  
WHERE supplier\_id >= 5000AND supplier\_id <= 5010;

You can use BETWEEN...AND operators to do the exact same thing in a more descriptive fashion, and using less typing to boot:

SELECT \*FROM airport

WHERE city BETWEEN 'London' AND 'Rome';

Both of the above queries produce exactly the same result set. The BETWEEN...AND operators specify an inclusive range (i.e. the endpoints of the range are included). To only select value outside of the range, the NOT keyword is used:

SELECT \*FROM airport

WHERE city NOT BETWEEN 'London' AND 'Rome';

The BETWEEN...AND operators work with all column datatypes, including number, date, and string types.

**Specify Lists in a WHERE Clause using IN**

When filtering the results of a query so that just a list of values is returned in the result set, you can either specify each item in the list in turn using OR:

SELECT \*FROM airport

WHERE city = 'Berlin'

OR city = 'Madrid'

OR city = 'Rome';

Or, you can use the IN operator to accomplish the same thing but with much less typing:

SELECT \*FROM airport

WHERE city IN ('Berlin', 'Madrid', 'Rome');

Note the parentheses around the list of values. We can also use the NOT operator to filter by values that are not in a list:

SELECT \*FROM airport

WHERE city NOT IN ('Berlin', 'Madrid', 'Rome');

**Sort Result Sets with ORDER BY**

The ORDER BY clause allows you to sort the records in your result set. The ORDER BY clause can only be used in SELECT statements. By default, each column is sorted in ascending order. To sort in descending order, type the keyword DESC after the column name.For instance,

SELECT supplier\_city, supplier\_state  
FROM suppliers  
WHERE supplier\_name = 'IBM'  
ORDER BY supplier\_city DESC, supplier\_state ASC;

This would return all records sorted by the supplier\_city field in descending order, with a secondary sort by supplier\_state in ascending order.

**Specify Wildcard String Matches in a WHERE Clause using LIKE**

Strings can be matched using wildcards in SQL just as one would use wildcards to match files in MS-DOS or UNIX. The wildcards are different, however. In SQL, the percent sign (%) matches "zero, one, or more characters" and the underscore (\_) matches "exactly one character".

The LIKE condition allows you to use wildcards in the where clause of an SQL statement. This allows you to perform pattern matching. The LIKE condition can be used in any valid SQL statement. The patterns that you can choose from are:

percent (%)allows you to match any string of any length (including zero length)

underscore (\_) allows you to match on a single character

SELECT \* FROM suppliers  
WHERE supplier\_name LIKE 'Hew%';

**Remember:** Patters are case sensitive.

**Pattern matching examples:**

‘Intro%’ matches any string beginning with “Intro”.

‘%Comp%’ matches any string containing “Comp” as a substring.

‘\_ \_ \_’ matches any string of exactly three characters.

‘\_ \_ \_ %’ matches any string of at least three characters.

**Renaming Result Set Columns using Column Aliases**

Just as tables can have aliases in queries, so too can columns. While table aliases are a technique to reduce typing, column aliases are strictly decorative. A column alias simply determines the name of the column in the result set--nothing else is affected in the query. To specify a column alias, type the alias immediately following the column name (or calculated expression) with the keyword AS between the name and alias (PostgreSQL insists on the AS keyword, unlike other RDBMSs):

SELECT r.rental\_id, apd.city AS departure\_city,

FROM rental r, airport apd

ORDER BY departure\_city, arrival\_city;

Column aliases are most useful when two or more column names would otherwise be displayed in the result set with the same name. Column aliases can also be used in an ORDER BY clause, as in the previous example, instead of the normal column name (particularly handy when you want to sort by a calculated expression) or in the ORDER BY clause used by the UNION of two or more queries. In the latter case, the column alias must be defined in the first SELECT statement in the query.

**Combine Multiple Result Sets with UNION**

The UNION query allows you to combine the result sets of 2 or more "select" queries. It removes duplicate rows between the various "select" statements.

Each SQL statement within the UNION query must have the same number of fields in the result sets with similar data types.

select field1, field2, . field\_n  
from tables  
UNION / UNION ALL  
select field1, field2, . field\_n  
from tables;

The UNION ALL query allows you to combine the result sets of 2 or more "select" queries. It returns all rows (even if the row exists in more than one of the "select" statements).

Each SQL statement within the UNION ALL query must have the same number of fields in the result sets with similar data types.

|  |
| --- |
| **SUBQUERY** |

A subquery is a query within a query. In Oracle, you can create subqueries within your SQL statements. These subqueries can reside in the WHERE clause, the FROM clause, or the SELECT clause.

**WHERE clause**

Most often, the subquery will be found in the WHERE clause. These subqueries are also called nested subqueries. For example:

select \* from all\_tables tabs

where tabs.table\_name in *(select cols.table\_name*

*from all\_tab\_columns cols*

*where cols.column\_name = 'SUPPLIER\_ID')*;

**FROM clause**

A subquery can also be found in the FROM clause. These are called inline views. For example:

select suppliers.name, subquery1.total\_amt

from suppliers,

*(select supplier\_id, Sum(orders.amount) as total\_amt*

*from orders*

*group by supplier\_id) subquery1,*

where subquery1.supplier\_id = suppliers.supplier\_id;

In this example, we've created a subquery in the FROM clause as follows:

(select supplier\_id, Sum(orders.amount) as total\_amt

from orders

group by supplier\_id) subquery1

This subquery has been aliased with the name subquery1. This will be the name used to reference this subquery or any of its fields.

**SELECT clause**

A subquery can also be found in the SELECT clause.For example:

select tbls.owner, tbls.table\_name,

*(select count(column\_name) as total\_columns*

*from all\_tab\_columns cols*

*where cols.owner = tbls.owner*

*and cols.table\_name = tbls.table\_name) subquery2*

from all\_tables tbls;

In this example, we've created a subquery in the SELECT clause as follows:

*(select count(column\_name) as total\_columns*

*from all\_tab\_columns cols*

*where cols.owner = tbls.owner*

*and cols.table\_name = tbls.table\_name) subquery2*

The subquery has been aliased with the name subquery2. This will be the name used to reference this subquery or any of its fields.

The trick to placing a subquery in the select clause is that the subquery must return a single value. This is why an aggregate function such as SUM, COUNT, MIN, or MAX is commonly used in the subquery.

**EXISTS / NOT EXISTS**

The EXISTS condition is considered "to be met" if the subquery returns at least one row.

SELECT columns  
FROM tables  
WHERE EXISTS ( subquery );

Let's take a look at a simple example. The following is an SQL statement that uses the EXISTS condition:

SELECT \*  
FROM suppliers  
WHERE EXISTS  
  (select \*  
    from orders  
    where suppliers.supplier\_id = orders.supplier\_id);

This select statement will return all records from the suppliers table where there is at least one record in the orders table with the same supplier\_id.

The EXISTS condition can also be combined with the NOT operator. For example,

SELECT \*  
FROM suppliers  
WHERE not exists (select \* from orders Where suppliers.supplier\_id = orders.supplier\_id);

This will return all records from the suppliers table where there are no records in the orders table for the given supplier\_id.

|  |
| --- |
| **DDL, DML** |

**Data Definition Language (DDL)**

DDL uses just five statements: CREATE, ALTER, DROP, GRANT, and REVOKE.

Data Definition Language (DDL) is all about creating, modifying and deleting relations.

Create table, drop table, ALTER TABLE,

**Data Manipulation Language**

DML uses just four statements: INSERT, UPDATE, DELETE, and SELECT.

Data Manipulation Language (DML)is all about adding, modifying, and removing data/tuples.

The four DML statements implement what is known as **CRUD** functionality: Create, Read, Update, and Delete. The actual DML statements use a couple of different names:

INSERT for Create and SELECT for Read, but the other two--UPDATE and DELETE--are the same.