USN | | | | | | | | | | |

CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Internal Test 2 – April 2018

Sub:	Database Management System				Sub Code:	17MCA23	Branch:	MCA
Date:	17/04/18	Duration:	90 min's	Max Marks: 50	Sem	II		OBE

Answer all FIVE FULL Questions

		MARKS	CO	RBT
Part I-1 (a)	. Write about embedded SQL. Explain with code how to retrieve multiple tuples in Embedded SQL	[10]	CO3	L2
	OR			
2 (a)	Explain aggregate or group functions in SQL with suitable example	[10]	CO3	L1
Part II-3 (a)	Discuss INSERT, DELETE and UPDATE statements in the modification of the database with examples	[7+3]	CO1	L2
(b)	Write short notes on : 1, NOT NULL 2, UNIQUE Constraint with examples.			
	OR			
4 (a)	What are DDL and DCL commands in SQL? Give an example of any one command from each.	[10]	CO3	L2
PartIII -5(a)	What are views in SQL? How is view created and dropped? what problems are associated with updating of views.	[10]	CO3	L2
	OR			
6(a)	Explain ON DELETE CASCADE, ON UPDATE CASCADE and CHECK clauses with examples	[10]	CO3	L2
Part IV-7(a)	Give a brief note on different types of joins with examples10M	[10]	CO3	L2
	OR			
8(a)	Explain how GROUP BY & ORDERED BY clause works? What is the difference between the WHERE & HAVING clause?	[10]	CO3	L2
Part V-9(a)	Write about authorization in SQL	[10]	CO3	L3
	OR			
10(a)	Consider the following structure of a database that keeps employees details and complete the queries given:	[10]	CO3	L3

Name	**Null?**	**Type**
EMPID	NOT NULL	NUMBER(5)
FNAME		VARCHAR2(25)
LNAME		VARCHAR2(25)
EMAIL		VARCHAR2(20)
PHONE		NUMBER(10)
DOJ		DATE
JOBID		VARCHAR2(4)
SAL		NUMBER(8)
MGRID		NUMBER(3)
DID		NUMBER(3)

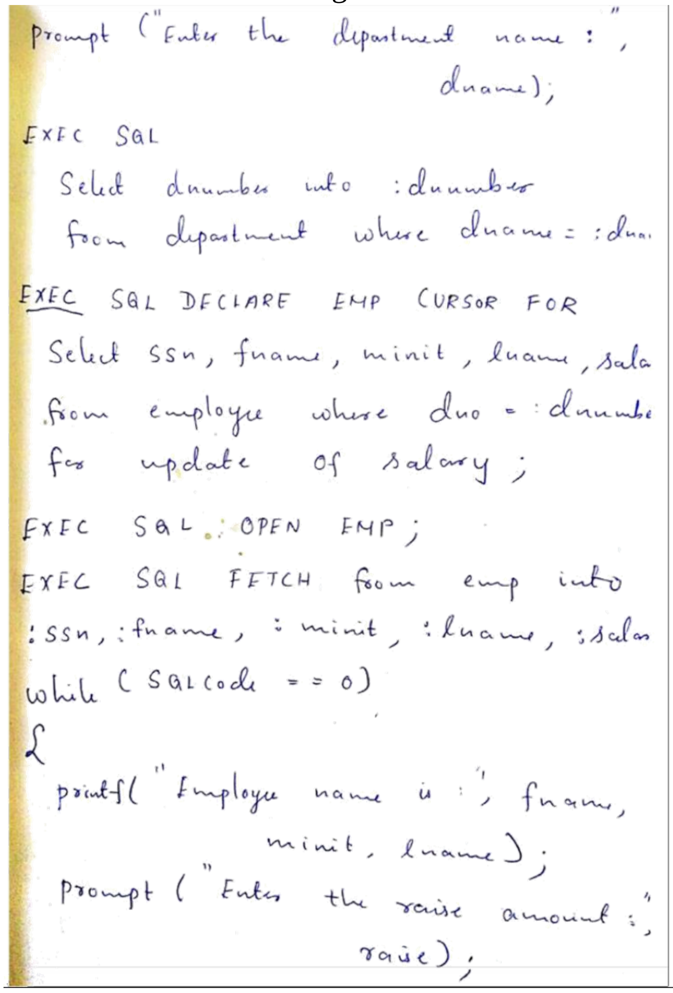
----------**Continuation**---------

1.Write a query to get unique department ID from employee table

2. Write a query to get all employee details from the employee table
order by first name, descending
3.Write a query to get the employee ID, names (first_name, last_name), salary
in ascending order of salary.
4.Write a query to get the number of jobs available in the employee table.
5.Write a query to select fname having ma.

1a. Write about embedded SQL. Explain with code how to retrieve multiple tuples in Embedded SQL. 10M

Specify the cursor using a DECLARE CURSOR statement.
Perform the query and build the result table using the OPEN statement.
Retrieve rows one at a time using the FETCH statement.
Process rows with the DELETE or UPDATE statements (if required).
Terminate the cursor using the CLOSE statement.



```
Prompt ("Enter the department name : ",
                               dname);
EXEC SQL
  Select dnumber into :dnumber
  from department where dname = :dna
EXEC SQL DECLARE EMP CURSOR FOR
  Select ssn, fname, minit, lname, sala
  from employee where dno = :dnumbe
  for update of salary ;
EXEC SQL OPEN EMP ;
EXEC SQL FETCH from emp into
:ssn, :fname, :minit, :lname, :salar
while ( SQLcode == 0)
{
  printf( "Employee name is :", fname,
                    minit, lname);
  Prompt ( "Enter the raise amount :",
                          raise);
EXEC SQL
  update EMPLOYEE set Salary = Salar
                               :raise
  where Current of EMP;

EXEC SQL FETCH from Emp into
  :ssn, :fname, :minit, :lname,
  :salary;
}
EXEC SQL CLOSE EMP;
```

Specifying Queries at Runtime using dynamic SQL:

```
//Program Segment E3:

EXEC SQL BEGIN DECLARE SECTION;
Varchar Sqlupdatestring [256];
EXEC SQL END DECLARE SECTION;
...
prompt ("Enter the update command",
   sqlupdatestring);
EXEC SQL PREPARE Sqlcommand FROM
        :Sqlupdatestring;
EXEC SQL EXECUTE Sql
```

2a. Explain aggregate or group functions in SQL with suitable example 10M

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate functions:

- Average: **avg**
- Minimum: **min**
- Maximum: **max**
- Total: **sum**
- Count: **count**

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.
To apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify 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.
"Find the average salary in each department."

select *dept name*, **avg** (*salary*) **as** *avg salary* **from** *instructor* **group by** *dept name*;

The specified aggregate is computed for each group.
When an SQL query uses grouping the attributes that appear in the select statement without being aggregated are those that are present in the group by clause. 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.

3a. Discuss INSERT, DELETE and UPDATE statements in the modification of the database with Examples 7M

Insertion
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. Similarly, tuples inserted must have the correct number of attributes.
Ex : There is a course CS-437 in the Computer Science department with title "Database Systems", and 4 credit hours.
insert into *course* **values** ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
We can delete only whole tuples; we cannot delete values on only particular attributes. SQL expresses a deletion by **delete from** *r* **where** *P*; where *P* represents a predicate and *r* represents a relation.
Deletion
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.
Delete all tuples in the *instructor* relation pertaining to instructors in the Finance department.
delete from *instructor* **where** *dept name*= 'Finance';
The **delete** request can contain a nested **select** that references the relation from which tuples are to be deleted.
Updates
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.
If a salary increase is to be paid only to instructors with salary of less than $70,000, we can write:
update *instructor*
set *salary* = *salary* * 1.05
where *salary* < 70000;
The **where** clause of the **update** statement may contain any construct legal in the **where** clause of the

select statement (including nested **select**s). As with **insert** and **delete**, a nested **select** within an **update** statement may reference the relation that is being updated. As before, SQL first tests all tuples in the relation to see whether they should be updated, and carries out the updates afterward.

3b. Write short notes on : 1, NOT NULL 2, UNIQUE Constraint with examples. 3M

Domain Constraints: Each attribute value must be either **null** (which is really a *non-value*) or drawn from the domain of that attribute. Note that some DBMS's allow you to impose the **not null** constraint upon an attribute, which is to say that that attribute may not have the (non-)value **null**.

NOT NULL

name varchar(20) not null
budget numeric(12,2) not null

The not null specification prohibits the insertion of a null value for the attribute.Any database modification that would cause a null to be inserted in an attribute declared to be not null generates an error diagnostic. There are many situations where we want to avoid null values.
SQL prohibits null values in the primary key of a relation schema. Thus, in our university example, in the *department* relation, if the attribute *dept name* is declared as the primary key for *department*, it cannot take a null value. As a result it would not need to be declared explicitly to be not null.

Unique Constraint

SQL also supports an integrity constraint:
unique (*Aj*1 , *Aj*2, . . . , *Ajm*)
The unique specification says that attributes *Aj*1 , *Aj*2, . . . , *Ajm* form a candidate key; that is, no two tuples in the relation can be equal on all the listed attributes. Candidate key attributes are permitted to be *null* unless they have explicitly been declared to be not null.

4a.What are DDL and DCL commands in SQL? Give an example of any one command from each. 10M

DDL (Data Definition Language)

DDL statements are used to alter/modify a database or table structure and schema. These statements handle the design and storage of database objects.
CREATE – create a new Table, database, schema
ALTER – alter existing table, column description
DROP – delete existing objects from database
DCL (Data Control Language)
DCL statements control the level of access that users have on database objects.
GRANT – allows users to read/write on certain database objects
REVOKE – keeps users from read/write permission on database objects

5a. What are views in SQL? How is view created and dropped? what problems are associated with updating of views. 10M

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.**
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*.
Ex: **create view** *faculty* **as select** *ID, name, dept name* **from** *instructor*;
The view relation is created whenever needed, on demand.
Views are usually implemented as follows.
When we define a view, the database system stores the definition of the view itself, rather than the result of evaluation of the query expression that defines the view. Wherever a view relation appears in a query,

it is replaced by the stored query expression. Thus, whenever we evaluate the query, the view relation is recomputed.
One view may be used in the expression defining another view.
In general, an SQL view is said to be **updatable** (that is, inserts, updates or deletes can be applied on the view) if the following conditions are all satisfied by the query defining the view:
- The **from** clause has only one database relation.
- The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
- Any attribute not listed in the **select** clause can be set to *null*; that is, it does not have a **not null** constraint and is not part of a primary key.
- The query does not have a **group by** or **having** clause.

6a.Explain ON DELETE CASCADE,ON UPDATE CASCADE and CHECK clauses with examples 10M

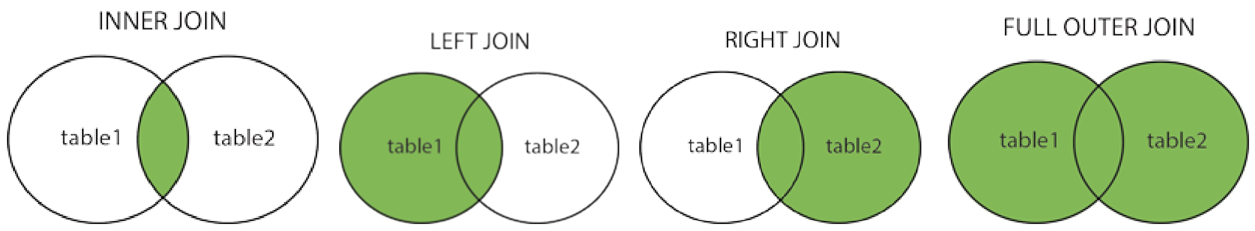
create table *course*(. . .**foreign key** (*dept name*) **references** *department*
on delete cascade
on update cascade,. . .);

Because of the clause **on delete cascade** associated with the foreign-key declaration, if a delete of a tuple in *department* results in this referential-integrity constraint being violated, the system does not reject the delete. Instead, the delete "cascades" to the *course* relation, deleting the tuple that refers to the department that was deleted. Similarly, the system does not reject an update to a field referenced by the constraint if it violates the constraint; instead, the system updates the field *dept name* in the referencing tuples in *course* to the new value as well.
SQL also allows the **foreign key** clause to specify actions other than **cascade**, if the constraint is violated: The referencing field (here, *dept name*) can be set to *null* (by using **set null** in place of **cascade**), or to the default value for the domain (by using **set default**).
If a cascading update or delete causes a constraint violation that cannot be handled by a further cascading operation, the system aborts the transaction. As a result, all the changes caused by the transaction and its cascading actions are undone.
Null values complicate the semantics of referential-integrity

7a. Give a brief note on different types of joins with examples. 10M

The different types of the JOINs in SQL:

- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Return all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Return all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Return all records when there is a match in either left or right table
- **NATURAL JOIN :** Returns all records only with the same value on those attributes that appear in both tables.
- **SELF JOIN** : Returns all records in the table where the table is joined with itself.



The **join . . .using** clause, which is a form of natural join that only requires values to match on specified attributes. The **on** condition allows a general predicate over the relations being joined.
This predicate is written like a **where** clause predicate except for the use of the keyword **on** rather than **where**. Like the **using** condition, the **on** condition appears at the end of the join expression.
Consider the following query, which has a join expression containing the **on**
condition.
select * **from** *student* **join** *takes* **on** *student.ID*= *takes.ID*;
The **outer join** operation works in a manner similar to the join operations we have already studied, but preserve those tuples that would be lost in a join, by creating tuples in the result containing null values.
There are in fact three forms of outer join:
• 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.
In contrast, the join operations studied earlier that do not preserve non matched tuples are called **inner join** operations, to distinguish them from the outer-join operations.
To compute the **left outer-join** operation as follows.
First, compute the result of the inner join as before.
Then, for every tuple *t* in the left-hand-side relation that does not match any tuple in the right-hand-side relation in the inner join, add a tuple *r* to the result of the join constructed as follows:
• The attributes of tuple *r* that are derived from the left-hand-side relation are filled in with the values from tuple *t*.
• The remaining attributes of *r* are filled with null values.
select * **from** *student* **natural left outer join** *takes*;
The **right outer join** is symmetric to the **left outer join**. Tuples from the right hand-side relation that do not match any tuple in the left-hand-side relation are padded with nulls and are added to the result of the right outer join. Thus, if we rewrite our above query using a right outer join and swapping the order in which we list the relations as follows:
select * **from** *takes* **natural right outer join** *student*;
The **full outer join** is a combination of the left and right outer-join types.
After the operation computes the result of the inner join, it extends with nulls those tuples from the left-hand-side relation that did not match with any from the right-hand side relation, and adds them to the result.
Similarly, it extends with nulls those tuples from the right-hand-side relation that did not match with any tuples from the left-hand-side relation and adds them to the result.

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**.
EX: **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**.

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.

8a. Explain how GROUP BY & ORDERED BY clause works? What is the difference between the WHERE & HAVING clause? 10M

To apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify 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.
Ex: "Find the average salary in each department."
select *dept name*, **avg** (*salary*) **as** *avg salary* **from** *instructor* **group by** *dept name*;

When an SQL query uses grouping, the only attributes that appear in the **select** statement without being aggregated are those that are present in the **group by** clause. 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.
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:

1. As was the case for queries without aggregation, the **from** clause is first evaluated to get a relation.
2. If a **where** clause is present, the predicate in the **where** clause is applied on the result relation of the **from** clause.
3. 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.
4. The **having** clause, if it is present, is applied to each group; the groups that do not satisfy the **having** clause predicate are removed.
5. 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.

To illustrate the use of both a **having** clause and a **where** clause in the same query, we consider the query "For each course section offered in 2009, find the average total credits (*tot cred*) of all students enrolled in the section, if the section had at least 2 students."
select *course id*, *semester*, *year*, *sec id*, **avg** (*tot cred*)
from *takes* **natural join** *student*
where *year* = 2009
group by *course id*, *semester*, *year*, *sec id*
having count (*ID*) >= 2;

9a. Write about authorization in SQL 10M

We may assign a user several forms of authorizations on parts of the database.

Authorizations on data include:

1. Authorization to read data.
2. Authorization to insert new data.
3. Authorization to update data.
4. Authorization to delete data.

Each of these types of authorizations is called a **privilege.** We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view. When a user submits a query or an update, the SQL implementation first checks if the query or update is authorized, based on the authorizations that the user has been granted. If the query or update is not authorized, it is rejected.

In addition to authorizations on data, users may also be granted authorizations on the database schema, allowing them, for example, to create, modify, or drop relations. A user who has some form of authorization may be allowed to pass on (grant) this authorization to other users, or to withdraw (revoke) an authorization that was granted earlier. In this section, we see how each of these authorizations can be specified in SQL. The ultimate form of authority is that given to the database administrator.

Granting and Revoking of Privileges

The SQL standard includes the privileges **select**, **insert**, **update**, and **delete**. The privilege **all privileges** can be used as a short form for all the allowable privileges. A user who creates a new relation is given all privileges on that relation automatically. The SQL data-definition language includes commands to grant and revoke privileges.

The **grant** statement is used to confer authorization.
The basic form of this statement is:
grant <privilege list> **on** <relation name or view name> **to** <user/role list>;
The *privilege list* allows the granting of several privileges in one command.

The **select** authorization on a relation is required to read tuples in the relation.
The following **grant** statement grants database users Amit and Satoshi **select** authorization on the *department* relation:
grant select on *department* **to** *Amit*, *Satoshi*;

This **grant** statement gives users Amit and Satoshi update authorization on the *budget* attribute of the *department* relation:
grant update (*budget*) **on** *department* **to** *Amit, Satoshi*;

The **insert** authorization on a relation allows a user to insert tuples into the relation. The **insert** privilege may also specify a list of attributes; any inserts to the relation must specify only these attributes, and the system either gives each of the remaining attributes default values (if a default is defined for the attribute) or sets them to *null*.
The **delete** authorization on a relation allows a user to delete tuples from a relation.
The user name **public** refers to all current and future users of the system.
Thus, privileges granted to **public** are implicitly granted to all current and future users.
To revoke an authorization, we use the **revoke** statement. It takes a form almost identical to that of **grant**:
revoke <privilege list> **on** <relation name or view name> **from** <user/role list>;
Thus, to revoke the privileges that we granted previously, we write
revoke select on *department* **from** *Amit*, *Satoshi*;
revoke update (budget) **on** *department* **from** *Amit*, *Satoshi*;
Revocation of privileges is more complex if the user from whom the privilege is revoked has granted the privilege to another user.

10a. Queries

1.Write a query to get unique department ID from employee table

select *from gemp where did in (select did from gemp group by did having count(*)=1);

2. Write a query to get all employee details from the employee table order by first name, descending

select *from gemp order by fname desc;

3.Write a query to get the employee ID, names (first_name, last_name), salary in ascending order of salary.

select empid, fname, lname, sal from gemp order by sal ;

4.Write a query to get the number of jobs available in the employee table.

select count(distinct(jobid)) "Number of Jobs" from gemp;

5.Write a query to select fname having ma.

select * from gemp where fname like '%ma%' ;