CS3481 DATABASE MANAGEMENT SYSTEMS LABORATORY

LIST OF EXPERIMENTS:

1. Create a database table, add constraints (primary key, unique, check, Not null), insert rows, update and delete rows using SQL DDL and DML commands.

2. Create a set of tables, add foreign key constraints and incorporate referential integrity.

3. Query the database tables using different ‘where’ clause conditions and also implement aggregate functions.

4. Query the database tables and explore sub queries and simple join operations.

5. Query the database tables and explore natural, equi and outer joins.

6. Write user defined functions and stored procedures in SQL.

7. Execute complex transactions and realize DCL and TCL commands.

8. Write SQL Triggers for insert, delete, and update operations in a database table.

9. Create View and index for database tables with a large number of records.

10. Create an XML database and validate it using XML schema.

11. Create Document, column and graph based data using NOSQL database tools.

12. Develop a simple GUI based database application and incorporate all the above mentioned features

13. Case Study using any of the real life database applications from the following list

a) Inventory Management for a EMart Grocery Shop

b) Society Financial Management

c) Cop Friendly App – Eseva

d) Property Management – eMall

e) Star Small and Medium Banking and Finance

● Build Entity Model diagram. The diagram should align with the business and functional goals stated in the application.

● Apply Normalization rules in designing the tables in scope.

● Prepared applicable views, triggers (for auditing purposes), functions for enabling enterprise grade features.

● Build PL SQL / Stored Procedures for Complex Functionalities, ex EOD Batch Processing for calculating the EMI for Gold Loan for each eligible Customer.

● Ability to showcase ACID Properties with sample queries with appropriate settings

Ex. No : 01 **DDL COMMANDS**
Date :

AIM:

To create a database and write SQL queries to retrieve information from the database.

DESCRIPTION:

Data Definition Language

DDL (Data Definition Language) statements are used to create, change the objects of a database. Typically a database administrator is responsible for using DDL statements or production databases in a large database system. The commands used are:

Create - It is used to create a table.

Alter - This command is used to add a new column, modify the existing column definition and to include or drop integrity constraint.

Drop - It will delete the table structure provided the table should be empty.

Truncate - If there is no further use of records stored in a table and the structure has to be retained, and then the records alone can be deleted.

Desc - This is used to view the structure of the table.

PROCEDURE:

Step 1: Create table by using create table command with column name, data type and size.

Step 2: Display the table by using desc command.

Step 3: Add any new column with the existing table by alter table command.

Step 4: Modify the existing table with modify command.

Step 5: Delete the records in files by truncate command.

Step 6: Delete the Entire table by drop command

Create Table
Syntax:

```
Create table tablename
(
column_name1 datatype(size),
column_name2 datatype(size),
column_name3 datatype(size),……);
```

Example:

SQL> Create table Student(Stud_name char(20), Stud_id varchar2(10), Stud_dept varchar2(20), Stud_age number(5));

Table created.

SQL> desc Student;

Name	Null? Type
STUD_NAME	CHAR(20)
STUD_ID	VARCHAR(10)
STUD_DEPT	VARCHAR(20)
STUD_AGE	NUMBER(5)

Alter Table
Syntax:

Alter table tablename add (column_name datatype(size));
Alter table tablename modify (column_name datatype(size));
Alter table tablename drop (column_name);

Example:

SQL> Alter table Student add (Stud_addr varchar2 (20));
Table altered.

SQL> desc Student;

Name	Null? Type
STUD _ NAME	CHAR(20)
STUD_ID	VARCHAR(10)
STUD_DEPT	VARCHAR(20)
STUD_AGE	VARCHAR(10)
STUD_ADDR	VARCHAR(20)

SQL> Alter table Student modify (Stud_age number(10));
Table altered.

SQL> desc Student;

Name	Null? Type
STUD _ NAME	CHAR(20)
STUD_ID	VARCHAR(10)
STUD_DEPT	VARCHAR(20)
STUD_AGE	NUMBER(10)
STUD_ADDR	VARCHAR(20)

SQL> Alter table Student drop (Stud_age number(10));
Table altered.

SQL> desc Student;

Name	Null? Type
STUD_NAME	CHAR(20)
STUD_ID	VARCHAR(10)
STUD_DEPT	VARCHAR(20)
STUD_ADDR	VARCHAR(20)

Truncate Table
Syntax:

Truncate table table_name;

Example:

SQL> Truncate table Student;

Table truncated.

SQL> desc Student

Name	Null? Type
STUD_NAME	CHAR(20)
STUD_ID	VARCHAR(10)
STUD_DEPT	VARCHAR(20)
STUD_AGE	NUMBER(10)
STUD_ADDR	VARCHAR(20)

Rename
Syntax

Alter table table_name rename new_table_name

SQL> alter table student rename student1;

SQL> desc student1;

Name	Null? Type
STUD_NAME	CHAR(20)
STUD_ID	VARCHAR(10)
STUD_DEPT	VARCHAR(20)
STUD_AGE	NUMBER(10)
STUD_ADDR	VARCHAR(20)

Drop Table
Syntax:

Drop table tablename;
Example:

SQL> Drop table Student1;

Table dropped.

SQL> desc Student1;

ERROR: ORA-04043: object Student1 does not exist

DML COMMANDS

AIM:

To Study and Practice Insertion, Deletion, Modifying, Altering, Updating and Viewing records based on conditions in RDBMS.

DESCRIPTION:

Data Manipulation Language

DML commands are the most frequently used SQL commands and is used to query and manipulate the existing database objects. Some of the commands are

Insert
Select
Update
Delete

PROCEDURE:

Step 1: Create table by using create table command.
Step 2: Insert values into the table
Step 3: Delete any records from the table
Step 4: Update any values in the table.
Step 5: Display the values from the table by using select command.

SQL> Create table Student(Stud_name char(20), Stud_id varchar2(10), Stud_ dept varchar2(20), Stud_age number(5));

Table created.

SQL> desc Student;

Name	Null?	Type
STUD_NAME		CHAR(20)
STUD_ID		VARCHAR(10)
STUD_DEPT		VARCHAR(20)
STUD_AGE		NUMBER(5)

Insert:
This is used to add one or more rows to a table. The values are separated by commas and the data types char and date are enclosed in apostrophes. The values must be entered in the same order as they are defined.

Syntax:
insert into tablename values(
‘&column_name1’,
‘&column_name2’, ‘
‘&column_name3’,…..);

Example:

SQL> Insert into Student1 values(‘&stud_name’, ‘&stud_id’, ‘&stud_dept’, ‘&stud_rollno’);

Insert into Student1 values (‘Ram’, ‘101’, ‘MECH’, ‘104’)

1 row created.

Insert into Student1 values (‘Vicky’, ‘102’, ‘EEE’, ‘105’)

1 row created.

Insert into Student1 values (‘Saddiq’, ‘102’, ‘CSE’, ‘101’)

1 row created.

Insert into Student1 values (‘David’, ‘104’, ‘EEE’, ‘103’)

1 row created.

Select Command:

It is used to retrieve information from the table. It is generally referred to as querying the table. We can either display all columns in a table or only specify column from the table.

Syntax:

Select * from table_name;

Example:

SQL> select * from Student1;

STUD_NAME	STUD_ID	STUD_DEPT	STUD_ROLL
Ram	101	MECH	104
Vicky	102	EEE	105
Saddiq	103	CSE	101
David	104	EEE	103

4 rows selected

Update Command:

It is used to alter the column values in a table. A single column may be updated or more than one column could be updated.

Syntax:

Update table_name set column_name=’value’ where condition;

Example:

SQL> Update Student1 set stud_id='109' where stud_name='Saddiq';

SQL> select * from Student1;

STUD_NAME	STUD_ID	STUD_DEPT	STUD_ROLLNO
Ram	101	MECH	104
Vicky	102	EEE	105
Saddiq	103	CSE	101
David	104	EEE	103

1 row updated.

Delete Command:

After inserting row in a table we can also delete them if required. The delete command consists of a from clause followed by an optional where clause. Syntax:

Delete from table_name where condition;

Example:

SQL> Delete from Student1 where stud_dept='CSE';
1 row deleted.

SQL> select * from Student1;

STUD_NAME	STUD_ID	STUD_DEPT	STUD_ROLLNO
Ram	101	MECH	104
Vicky	102	EEE	105
David	104	EEE	103

RESULT:

Thus the Insertion, Deletion, Modifying, Altering, Updating and Viewing records based on conditions using SQL commands were executed and verified successfully.

Ex. No.: 02

Date : **Foreign Key Constraints**

Aim:

To study and practice to create table, add foreign key constraints and incorporate referential integrity.

Description:

A referential integrity constraint is also known as **foreign key constraint**. A foreign key is a key whose values are derived from the Primary key of another table.

The table from which the values are derived is known as **Master or Referenced** Table and the Table in which values are inserted accordingly is known as **Child or Referencing** Table,

In other words, we can say that the table containing the **foreign key** is called the **child table**, and the table containing the **Primary key/candidate key** is called the **referenced or parent table**.

Procedure:

Step 1: Create the master or referenced table with required fields.
Step 2: Create the child table.
Step 3: Create the primary key in master table.
Step 4: Apply the insert and delete constrains.

CONSTRAINTS:

1) Primary key
2) Foreign key/references
3) Check
4) Unique
5) Not null
6) Null
7) Default

CONSTRAINTS CAN BE CREATED IN THREE WAYS:

1) Column level constraints
2) Table level constraints
3) Using DDL statements-alter table command

OPERATION ON CONSTRAINT:

i) ENABLE
ii) DISABLE
iii) DROP

NOT NULL:

Syntax:

Create table tablename(
fieldname1 datatype(constraint)not null,

fieldname2 datatype,

…….
fieldnamen datatype);

Example:

SQL> create table notnull (eno varchar(10) not null, ename varchar(10),esalary number(20));

Table created
SQL>insert into notnull values(‘1’,’abdul’,’20000’)
1 row created.

SQL>insert into notnull values(‘’,’raj’,’30000’)

*

ERROR at line 1:

ORA-01400: cannot insert NULL into (“SCOTT”.”NOTNULL”.”ENO”)

CHECK:

Check constraint specify conditions that each tuple must satisfy.

Syntax:

Create table tablename(
Fieldname1 datatype(constraint),
Fieldname2 datatype,
…….
Fieldname3 datatype);

Example:

SQL> create table con (empid decimal(10) not null, empname varchar(20),empsalary decimal(10), check(empsalary>10000));

SQL>insert into con values (‘1’,’kumar’,’20000’)
1 row created

SQL>insert into con values(‘2’,’raja’,’9000’)

*

ERROR at line 1:

ORA-02290: check constraint (SCOTT.SYS_C0010283) violated

UNIQUE:

Used to set unique constraint to the specified column name which will not allow redundant values

Syntax:

Create table tablename(
fieldname1 datatype(constraint)unique,
fieldname2 datatype,

…….
Fieldname3 datatype);

Example:

SQL> create table conn(eno varchar(10) unique, ename varchar(20));

Table created.
SQL> insert into conn values(‘1’,’hello’)
1 row created.

SQL>insert into conn values(‘1’,’hi’)

*

ERROR at line 1:

ORA-00001: unique constraint (SCOTT.SYS_C0010285) violated

PRIMARY KEY:
Primary key is a constraint for both unique and not null.

Syntax:

Create table tablename(
Fieldname1 datatype(constraint)primary key,
fieldname2 datatype,
…….
Fieldname3 datatype);

Example:

SQL> create table con(empid varchar(10),empname varchar(20) primary key);
Table created.

ADDING CONSTRIANT:

Used to set any constraint to the specified column at the last by specifying the constraint type and field name.

Syntax:

Create table tablename(
Fieldname1 datatype(constraint),

fieldname2 datatype,
constraint constraintname constrainttype(fieldname));

Example:

SQL> create table con(empid varchar(10),empname varchar(10),constraint c1 primary key(empid));

Table created.
SQL> insert into con values (‘1’,’anand’)

SQL>insert into con values (‘1’,’vijay’)

*

ERROR at line 1:

ORA-00001: unique constraint (SCOTT.C1) violated

ADD CONSTRAINT (ALTER)

Used to set the constraint for the table already created by using alter command.

Syntax:

Alter table tablename add constraint constraintname (fieldname)datatype,primary key.

Example:

SQL> create table con(empid varchar(10),empname varchar(10));

Table created.

SQL> alter table con add constraint c1 primary key (empid);

Table altered.

SQL> desc con;

Name	Null? Type
EMPID	NOT NULL VARCHAR(10)
EMPNAME	VARCHAR(10)

DROP CONSTRAINT:

Used to drop the constraint.

Syntax:

Alter table table_name drop constraint constraint_name.

Example:

SQL> alter table con drop constraint c1;

Table altered.

SQL> desc con;

Name	Null? Type
EMPID	VARCHAR(10)
EMPNAME	VARCHAR(10)

REFERENTIAL INTEGRITY:

Used to refer the primary key of the parent table from the child table.

Syntax:

a)Create table tablename(
Fieldname1 datatype primary key,
fieldname2 datatype,
…….
Fieldname3 datatype);

b) Create table tablename(Fieldname1 datatype references,
Parent tablename(fieldname)
…….
Fieldname n datatype);

Example:

SQL> create table parent(eno varchar(10),ename varchar(10) primary key);
Table created.

SQL>insert into parent values (‘1’,’ajay’)
1 row created.
SQL>insert into parent values (‘2’,’bala’)
1 row created.

SQL> create table child (eno varchar(10),ename varchar(10) references parent(ename));

Table created.

SQL>insert into child values (‘1’,’ajay’)
1 row created.

SQL>insert into child values (‘2’,’balaji’)

ERROR at line 1:

ORA-02291: integrity constraint (SCOTT.SYS_C0010290) violated - parent key not

Found

ON DELETE CASCADE:
The changes done in parent table is reflected in the child table when references are made.

Syntax:

Create table tablename(
Fieldname1 datatype references,
Parent tablename(fieldname),
On delete cascade);

Example:

SQL> create table parent(eno varchar(10),ename varchar(10) primary key);
Table created.

SQL>insert into parent values (‘1’,’a’)

1 row created.

SQL> create table child(eno varchar(10),ename varchar(10) references parent(ename) on delete cascade);

Table created.

SQL> insert into child values (‘2’,’a’)

1 row created.
SQL> select * from parent;

ENO ENAME
1 a

SQL> select * from child;

ENO ENAME
2 a

SQL> delete from parent where eno=1;

1 row deleted.

SQL> select * from parent;

no rows selected

SQL> select * from child;

no rows selected

Result

Thus the various key constraints based on foreign key where written and executed successfully.

Ex.No. : 03

Date : **Where and Aggregate Functions**

Aim:

To study and execute various database queries using where clause and aggregate functions.

Description:

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

Where clause's

Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>	Not equal.(Note: In some versions of SQL this operator may be written as !=)
BETWEEN	Between a certain range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

The WHERE clause is not only used in SELECT statements, it is also used in UPDATE, DELETE, etc.!

Equal

SQL> select * from table_name where field=condition

Example

SQL> select * from student1 where stud_rollno=101;

STUD_NAME	STUD_ID	STUD_DEPT	STUD_ROLLNO
Saddiq	103	CSE	101

Greater Than

SQL> select * from student1 where stud_rollno >101;

STUD_NAME	STUD_ID	STUD_DEPT	STUD_ROLLNO
Ram	101	MECH	104
Vicky	102	EEE	105
David	104	EEE	103

Less Than

SQL> select * from student1 where stud_rollno <105;

STUD_NAME	STUD_ID	STUD_DEPT	STUD_ROLLNO
Ram	101	MECH	104
Saddiq	103	CSE	101
David	104	EEE	103

Between

SQL> select * from student1 where stud_rollno between 103 AND 105;

STUD_NAME	STUD_ID	STUD_DEPT	STUD_ROLLNO
Ram	101	MECH	104
Vicky	102	EEE	105
David	104	EEE	103

Like

Syntax

SELECT *column1, column2, ...*
FROM *table_name*
WHERE *columnN* LIKE *pattern*;

LIKE Operator	**Description**
WHERE CustomerName LIKE 'a%'	Finds any values that start with "a"
WHERE CustomerName LIKE '%a'	Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%'	Finds any values that start with "a" and are at least 2 characters in length
WHERE CustomerName LIKE 'a__%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

Example

SQL > Select * from student1 where stud_name like ‘d%’;

STUD_NAME	STUD_ID	STUD_DEPT	STUD_ROLLNO
David	104	EEE	103

Other Where Clauses

Union:

The UNION operator is used to combine the result-set of two or ore SELECT statements.

Every SELECT statement within UNION must have the same number of columns
The columns must also have similar data types
The columns in every SELECT statement must also be in the same order

Syntax:

select column_name(s) from table1
union
select column_name(s) from table2*;*

SQL> select subject from student union select subject from staff order by subject;

Union with where

Syntax

Select <fieldlist> from <tablename1> where (condition)
union
select<fieldlist> from<tablename2> where (condition);

Example

SQL> SELECT dept, subject FROM student
WHERE subject='DBMS'
UNION
SELECT dept, subject FROM staff
WHERE subject='DBMS'
ORDER BY City;

Union All

select column_name(s) from table1
union all
select column_name(s) from table2;

Example

select subject from student
union all
select subject from staff
order by subject;

Intersect:

Syntax:

Select <fieldlist> from <tablename1> where (condition) intersect select<fieldlist> from<tablename2>

where (condition);

Example

SQL> select stud_id from student intersect select staff_id from staff;

In:

The IN operator allows you to specify multiple values in a WHERE clause.
The IN operator is a shorthand for multiple OR conditions.

Syntax:

Select <fieldlist> from <tablename1> where (condition) in select<fieldlist> from<tablename2> where

(condition);

Example

SQL>select * from student1 where stud_dept in ('cse', 'mech');

STUD_NAME	STUD_ID	STUD_DEPT	STUD_ROLLNO
Ram	101	MECH	104
Saddiq	103	CSE	101

SQL>select * from customers where stud_dept not in ('cse', 'mech', 'it');

STUD_NAME	STUD_ID	STUD_DEPT	STUD_ROLLNO
Vicky	102	EEE	105
David	104	EEE	103

Not like:

Syntax:

Select <fieldlist> from <tablename> where <fieldname> not like <expression>;

All:

The ALL operator:

returns a boolean value as a result
returns TRUE if ALL of the subquery values meet the condition
is used with SELECT, WHERE and HAVING statements

ALL means that the condition will be true only if the operation is true for all values in the range.

Syntax:

Select <fieldlist> from <tablename1>where <fieldname> all Select <fieldlist> from <tablename2>

where (condition);

Example

SQL> select stud_name from student where stud_id = all (select subject_id from subject where sem = 4);

Any:

The ANY operator:

returns a Boolean value as a result
returns TRUE if ANY of the sub query values meet the condition

ANY means that the condition will be true if the operation is true for any of the values in the range.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY
  (SELECT column_name
```

FROM table_name
WHERE condition*);*

The operator must be a standard comparison operator (=, <>, !=, >, >=, <, or <=).

Example

SQL> select stud_name from student where stud_id = any (select subject_id from subject where sem = 4);

The above SQL statement lists the student name if it finds ANY records in the subject table has sem equal to 4 (this will return TRUE if the sem column having value 4)

Aggregate Functions

MySQL's aggregate function is used to perform calculations on multiple values and return the result in a single value like the average of all values, the sum of all values, and maximum & minimum value among certain groups of values. We mostly use the aggregate functions with SELECT statements in the data query languages.

Aggregate Function	**Descriptions**
count()	It returns the number of rows, including rows with NULL values in a group.
sum()	It returns the total summed values (Non-NULL) in a set.
average()	It returns the average value of an expression.
min()	It returns the minimum (lowest) value in a set.
max()	It returns the maximum (highest) value in a set.
groutp_concat()	It returns a concatenated string.
first()	It returns the first value of an expression.
last()	It returns the last value of an expression.

Sql > create table student(rollno decimal, sname varchar(15), mark1 decimal, mark2 decimal);
Table created
Sql> insert into student values(101, ‘kareem’,95,90);
Sql> insert into student values(102, ‘kaasim’,92,97);
Sql> insert into student values(103, ‘ram’,85,95);
Sql> insert into student values(104, ‘sai’,93,91);

Sql> select * from student;

Rollno	sname	mark1	mark2
101	kareem	95	90
102	kaasim	92	97
103	ram	85	95
104	sai	93	91

Count

The COUNT () function returns the number of rows that matches a specified criterion.

Syntax
SELECT COUNT (column_name) FROM table_name WHERE condition;
Example
SELECT COUNT (rollno) FROM student;
4

Sum

The SUM () function returns the total sum of a numeric column.

Syntax
SELECT SUM(column_name) FROM table_name WHERE condition;
Example
SELECT SUM(mark1) FROM student;
365

SELECT SUM(mark1) FROM student WHERE mark2>=95;
192

Average

The AVG() function returns the average value of a numeric column.

Syntax
SELECT AVG(column_name) FROM table_name WHERE condition;
Example
SELECT AVG(mark1) FROM student;
91

SELECT AVG(mark2) FROM student WHERE mark2>=95;
96

Min

The MIN() function returns the smallest value of the selected column.

SELECT MIN(column_name) FROM table_name WHERE condition;
Example
SELECT MIN(mark1) FROM student;
85

Max

The MAX() function returns the largest value of the selected column.

SELECT MAX(column_name) FROM table_name WHERE condition;
Example
SELECT MAX(mark1) FROM student;
95

Result

Thus the where clause function quires and aggregate function quires are executed successfully.

Ex.no. : 04
Date :

Sub Quires and Join Operations

Aim :

To implement and execute simple, nested, sub & join operation queries in mysql database.

Simple Queries

The SQL SELECT DISTINCT Statement

The SELECT DISTINCT statement is used to return only distinct (different) values. Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.

Syntax

SELECT DISTINCT *column1, column2, ...*FROM *table_name*;

Example

SELECT DISTINCT STU_DEPT FROM student1;

STU_DEPT
CSE
EEE
MECH

The SQL AND, OR and NOT Operators

The WHERE clause can be combined with AND, OR, and NOT operators.

The AND and OR operators are used to filter records based on more than one condition:

- The AND operator displays a record if all the conditions separated by AND are TRUE.
- The OR operator displays a record if any of the conditions separated by OR is TRUE.

The NOT operator displays a record if the condition(s) is NOT TRUE

AND Syntax

SELECT *column1, column2, ...*
FROM *table_name*
WHERE *condition1* AND *condition2* AND *condition3* ..

Example

SELECT * FROM student1 WHERE stud_id=101 AND stud_dept='mech';

STUD_NAME	STUD_ID	STUD_DEPT	STUD_ROLLNO
Ram	101	MECH	104

OR Syntax

SELECT *column1, column2, ...*
FROM *table_name*
WHERE *condition1* OR *condition2* OR *condition3 ...*;

Example
SELECT * FROM student1 WHERE stud_id=101 OR stud_dept='EEE';

STUD_NAME	STUD_ID	STUD_DEPT	STUD_ROLLNO
Ram	101	MECH	104

NOT Syntax

SELECT *column1*, *column2*, ...
FROM *table_name*
WHERE NOT *condition*;

Example
SELECT * FROM student WHERE NOT stud_id=101;

STUD_NAME	STUD_ID	STUD_DEPT	STUD_ROLLNO
Vicky	102	EEE	105
David	104	EEE	103

The SQL ORDER BY Keyword

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.

ORDER BY Syntax

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

Example
SELECT * FROM student1 ORDER BY STUD_ID ;

STUD_NAME	STUD_ID	STUD_DEPT	STUD_ROLLNO
Ram	101	MECH	104
Vicky	102	EEE	105
Saddiq	103	CSE	101
David	104	EEE	103

SELECT * FROM student1 ORDER BY STUD_ID DESC;

STUD_NAME	STUD_ID	STUD_DEPT	STUD_ROLLNO
David	104	EEE	103
Saddiq	103	CSE	101
Vicky	102	EEE	105
Ram	101	MECH	104

Subqueries

A MySQL sub query is a query nested within another query such as SELECT, INSERT, UPDATE or DELETE. In addition, a MySQL sub query can be nested inside another sub query.

A MySQL sub query is called an inner query while the query that contains the sub query is called an outer query. A sub query can be used anywhere that expression is used and must be closed in parentheses.

Example SubQueries

1. SELECT lastName, firstName FROM employees WHERE officeCode IN (SELECT officeCode FROM offices WHERE country = 'USA');

In this example:

- The sub query returns all office codes of the offices located in the USA.
- The outer query selects the last name and first name of employees who work in the offices whose office codes are in the result set returned by the sub query.

2. Select max(sid) from classa where sid <(select max(sid) from classa)

SQL Joins

Here are 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

Note : To perform join operation we need two different tables.

Sql> select * from student;

Rollno	sname	mark1	mark2
101	kareem	95	90
102	kaasim	92	97
103	ram	85	95
104	sai	93	91

Sql > select * from sports

Rollno	sname	sdept	game
101	kareem	CSE	cricket
104	sai	ECE	football
105	ravi	IT	cricket
107	fizal	CSE	chess

Inner Join

The INNER JOIN keyword selects records that have matching values in both tables.

Syntax

SELECT *column_name(s)*
FROM *table1*
INNER JOIN *table2*
ON *table1.column_name = table2.column_name*;

Example

SELECT student.name, student.mark1, sports.game FROM student INNER JOIN sports ON student.rollno=sports.rollno;

sname	mark1	game
kareem	95	cricket
sai	93	football

Left Join

The LEFT JOIN keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match.

Syntax

SELECT *column_name(s)*
FROM *table1*
LEFT JOIN *table2*
ON *table1.column_name = table2.column_name*;

Example

SELECT student.name, student.mark1, sports.game FROM student LEFT JOIN sports ON student.rollno=sports.rollno;

sname	mark1	game
kareem	95	cricket
kaasim	92	Null
ram	85	Null
sai	93	football

RIGHT JOIN Keyword

The RIGHT JOIN keyword returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.

Syntax

SELECT *column_name(s)*
FROM *table1*
RIGHT JOIN *table2*

ON *table1.column_name = table2.column_name*;

Example

SELECT student.name, student.mark1, sports.game FROM student RIGHT JOIN sports ON student.rollno=sports.rollno;

sname	mark1	game
kareem	95	cricket
sai	93	football
Null	Null	cricket
Null	Null	chess

FULL OUTER JOIN Keyword

The FULL OUTER JOIN keyword return all records when there is a match in either left (table1) or right (table2) table records.

Note: FULL OUTER JOIN or FULL JOIN is not directly performed in sql so we can achive it by union operation of left join and right join.

Syntax

SELECT *column_name(s)*
FROM *table1*
FULL OUTER JOIN *table2*
ON *table1.column_name = table2.column_name*;

If the above syntax is not working then we can go with union operation.

SELECT *column_name(s)*
FROM *table1*
LEFT JOIN *table2*
ON *table1.column_name = table2.column_name*;
Union
SELECT *column_name(s)*
FROM *table1*
RIGHT JOIN *table2*
ON *table1.column_name = table2.column_name*;

Example

SELECT student.name, student.mark1, sports.game FROM student LEFT JOIN sports ON student.rollno=sports.rollno
Union
SELECT student.name, student.mark1, sports.game FROM student RIGHT JOIN sports ON student.rollno=sports.rollno

Result

Thus the SQL sub queries, nested queries and various join operation queries are written and executed successfully.

Ex. No. : 05
Date :

Natural, equi and outer joins

Aim:

To study and execute SQL natural join, equi join and outer joins.

Procedure:

Natural join is an SQL join operation that creates join on the base of the common columns in the tables. To perform natural join there must be one common attribute(Column) between two tables. Natural join will retrieve from multiple relations.

It works in three steps.

- Natural join is an SQL join operation that creates join on the base of the common columns in the tables.
- To perform natural join there must be one common attribute(Column) between two tables.
- Natural join will retrieve from multiple relations. It works in three steps.

Tables

Student

Roll	sname	dept
101	aaa	cse
105	eee	it
102	bbb	ece
103	ccc	eee
104	ddd	cse
105	eee	it

Game

Gid	gname	roll
1	cricket	101
2	volley ball	102
3	cricket	104
4	carom	106
5	chess	107

Natural Join Syntax

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

Example

```
Select * from student NATURAL JOIN game;
```

Roll	sname	dept	gid	game
101	aaa	cse	1	cricket
102	bbb	ece	2	volly ball
104	ddd	cse	3	cricket

Equi join:

- EQUI JOIN creates a JOIN for equality or matching column(s) values of the relative tables.
- EQUI JOIN also create JOIN by using JOIN with ON and then providing the names of the columns with their relative tables to check equality using equal sign (=).

Syntax

SELECT column_list
FROM table1, table2....
WHERE table1.column_name =
table2.column_name;

Example

SELECT student.roll,student.sname,game.gname FROM student,game WHERE student.roll=game.roll;
Or
SELECT student.roll,student.sname,game.gname FROM student JOIN game ON
student.roll=game.roll; Roll sname game
101 aaa cricket
102 bbb volly ball
104 ddd cricket

Non Equi Join :

NON EQUI JOIN performs a JOIN using comparison operator other than equal(=) sign like >, <, >=, <= with conditions.

Syntax

SELECT column_list
FROM table1, table2....
WHERE table1.column_name >
table2.column_name;

Example

SELECT student.roll,student.sname,game.gname FROM student,game WHERE student.roll>game.roll;

Roll	dept	game
105	eee	cricket
105	eee	volley ball
105	eee	cricket
102	bbb	cricket
103	ccc	cricket
103	ccc	volley ball
104	ddd	cricket
104	ddd	volley ball
105	eee	cricket
105	eee	volley ball
105	eee	cricket

Full outer join

The FULL OUTER JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records. Full outer join and full join are same

Syntax

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name= table2.column_name
WHERE condition;
```

Example

select student.roll,student.sname,game.gname from student left join game on student.roll=game.roll union
select student.roll,student.sname,game.gname from student right join game on student.roll=game.roll;

Roll	sname	game
101	aaa	cricket
105	eee	Null
102	bbb	volley ball
103	ccc	Null
104	ddd	cricket
Null	Null	carom
Null	Null	chess

Result:

Thus the natural join, equi join and outer join queries are written and executed successfully.

Ex.No. :06
Date :

Functions and Stored Procedures

Aim:

To Write a program using procedures and functions

MySQL Stored Function

A stored function is a special kind stored program that returns a single value. You use stored functions to encapsulate common formulas or business rules that are reusable among SQL statements or stored programs.

Different from a stored procedure, you can use a stored function in SQL statements wherever an expression is used. This helps improve the readability and maintainability of the procedural code.

syntax

The following illustrates the simplest syntax for creating a new stored function:

```
CREATE FUNCTION function_name(parameter 1,parameter 2,…)
RETURNS datatype
[NOT] DETERMINISTIC
Statements
```

Example

Function to concatenate two strings

```
USE `sample1`;
DROP function IF EXISTS `funcon`;
DELIMITER $$
USE `sample1`$$
CREATE DEFINER=`root`@`localhost` FUNCTION `funcon`(s CHAR(20)) RETURNS char(50)
CHARSET utf8mb4
DETERMINISTIC
BEGIN
RETURN CONCAT('Hello, ',s,'!!');
RETURN 1;
END$$
DELIMITER ;
```

Executing function

```
select funcon('world');
```

```
# funcon('world')
'Hello, world!!'
```

Stored procedure

MySQL stored procedure using CREATE PROCEDURE statement. In addition, we will show you how to call stored procedures from SQL statements.

syntax

```
DELIMITER //
CREATE PROCEDURE GetAllProducts()
BEGIN
SELECT * FROM products;
END //
DELIMITER ;
```

Example

```
create table cus(cid integer,cname char(20),address varchar(75),salary int,post varchar(20));

insert into cus values(1,'aa','77,anna salai,arcot',10000,'clerk');
insert into cus values(3,'bb','01,anna salai,chennai',15000,'staff');
insert into cus values(2,'cc','25,rajaji nagar,banglore',15000,'staff');
insert into cus values(4,'dd','02,mettu street,kochin',10000,'secretary');
insert into cus values(5,'ee','21,north street,mumbai',15000,'manager');

select* from cus;
```

Cid	cname	address	salary	post
1	aa	77,anna salai,arcot	10000	clerk
3	bb	01,anna salai,chennai	15000	staff
2	cc	25,rajaji nagar,banglore	15000	staff
4	dd	02,mettu street,kochin	10000	secretary
5	ee	21,north street,mumbai	15000	manager

Creating stored procedure

```
USE `sample1`;
DROP procedure IF EXISTS `new_pro`;
DELIMITER $$
USE `sample1`$$
CREATE DEFINER=`root`@`localhost` PROCEDURE `new_pro`()
BEGIN
UPDATE cus
SET salary = salary + 500;
END$$
DELIMITER ;
```

Executing stored procedure

call new_pro;

5 rows effected

SQL>select* from cus;

Cid	cname	address	salary	post
1	aa	77,anna salai,arcot	10500	clerk
3	bb	01,anna salai,chennai	15500	staff
2	cc	25,rajaji nagar,banglore	15500	staff
4	dd	02,mettu street,kochin	10500	secretary
5	ee	21,north street,mumbai	15500	manager

Result :

Thus the SQL functions and procedures are written and executed successfully.

Ex.No. :07
Date :

DCL and TCL Commands

Aim:

To study and execute various Data Control Language and Transaction Control Language commands in SQL.

Procedure:

1: Start
2: Create the table with its essential attributes.
3: Insert the record into table
4: Execute the DCL commands GRANT and REVOKE
5: Execute the TCL commands COMMIT, SAVEPOINT and ROLLBACK.
6: Stop

DCL Commands.

DCL includes commands such as GRANT and REVOKE which mainly deal with the rights, permissions, and other controls of the database system.

GRANT:

This command gives users access privileges to the database. For this first we have to create user.

CREATE USER Statement

MySQL allows us to specify which user account can connect to a database server. The user account details in MySQL contains two information – username and host from which the user is trying to connect in the format username@host-name.
If the admin user is connecting through localhost then the user account will be admin@localhost.

MySQL stores the user account in the user grant table of the mysql database.

The CREATE USER statement in MySQL allows us to create new MySQL accounts or in other words, the CREATE USER statement is used to create a database account that allows the user to log into the MySQL database.

Syntax;

CREATE USER user_account IDENTIFIED BY password;

NOTE: in our system user is root@localhost

We have already learned about how to create users in MySQL using MySQL | create user statement. But using the Create User Statement only creates a new user but does not grant any privileges to the user account. Therefore to grant privileges to a user account, the GRANT statement is used.

Syntax:

GRANT privileges_names ON object TO user;

Parameters Used in Grant Command

privileges_name: These are the access rights or privileges granted to the user.

object: It is the name of the database object to which permissions are being granted. In the case of granting privileges on a table, this would be the table name.

user: It is the name of the user to whom the privileges would be granted.

Various privileges used are, SELECT, INSERT, DELETE, INDEX, UPDATE, CREATE, ALTER, DROP, GRANT.

1. Granting SELECT Privilege to a User in a Table:

To grant Select Privilege to a table named “users” where User Name is root, the following GRANT statement should be executed.

GRANT SELECT ON Users TO 'root'@'localhost;

2. Granting more than one Privilege to a User in a Table:

To grant multiple Privileges to a user named “root” in a table “users”, the following GRANT statement should be executed.

GRANT SELECT, INSERT, DELETE, UPDATE ON Users TO 'root'@'localhost;

3. Granting All the Privilege to a User in a Table:

To Grant all the privileges to a user named “root” in a table “users”, the following Grant statement should be executed.

GRANT ALL ON Users TO 'root'@'localhost;

4. SQL Grant command is specifically used to provide privileges to database objects for a user. This command also allows users to grant permissions to other users too.

Syntax:

grant privilege_name on object_name

to {user_name | public | role_name}

Example

grant insert,

select on accounts to Ram

REVOKE:

This command withdraws the user's access privileges given by using the GRANT command.

Revoke command withdraw user privileges on database objects if any granted. It does operations opposite to the Grant command. When a privilege is revoked from a particular user U, then the privileges granted to all other users by user U will be revoked.

Syntax:

REVOKE privilege_name on object_name

from {user_name | public | role_name}

Example

REVOKE insert,

select on accounts from Ram

TCL (Transaction Control Language)

Transaction Control Language(TCL) commands are used to manage transactions in database.
These are used to manage the changes made by DML statements.
It also allows statements to be grouped together into logical transactions.

TCL Commands are

- Commit

Commit command is used to permanently save any transaction into database.Following is Commit command's syntax,
commit;

- Rollback

This command restores the database to last commited state. It is also use with savepoint command to jump to a savepoint in a transaction. Following is Rollback command's syntax,
rollback to savepoint-name;

- Savepoint

savepoint command is used to temporarily save a transaction so that you can rollback to that point whenever necessary.
Following is savepoint command's syntax,
savepoint savepoint-name;

Example

Create the table class with the ID & NAME attributes Apply TCL commands and show the result.

SQL> CREATE TABLE CLASS(id int,sname varchar(20));

SQL> insert into class values(1,'john');

SQL> insert into class values(2,'raj');

SQL> insert into class values(3,'rahman');

SQL>Select * from class

Id	sname
1	john
2	raj
3	rahman

Let’s use some SQL queries on the above table and see the results.

SQL> INSERT into class values(5,'Rahul');
commit;
SQL> UPDATE class set name='abhijit' where id='5';
savepoint A;
SQL> INSERT into class values(6,'Chris');
savepoint B;
SQL> INSERT into class values(7,'Bravo');
savepoint C;
SQL> SELECT * from class;

The resultant table will look like,

Id	sname
1	john
2	raj
3	rahman
5	abhijit
6	Chris
7	Bravo

Now rollback to savepoint B
rollback to B;
SELECT * from class;
The resultant table will look like

Id	sname
1	john
2	raj
3	rahman
5	abhijit
6	Chris

Now rollback to savepoint A
rollback to A;
SELECT * from class;

The result table will look like

Id	sname
1	john
2	raj
3	rahman
5	abhijit

Result:

Thus the Data Manipulation Language commands (insert, update, delete, retrieve) and TCL commands are studied and executed successfully and output was verified.

Ex. No.:08
Date :

Triggers

Aim:

To execute programs for insert, delete, and update operations in a database table using triggers.

MySQL trigger

A MySQL trigger is a stored program (with queries) which is executed automatically to respond to a specific event such as insertion, updation or deletion occurring in a table.

In order to create a new trigger, you use the CREATE TRIGGER statement. The following illustrates the syntax of the CREATE TRIGGER statement:

```
CREATE TRIGGER trigger_name trigger_time trigger_event
ON table_name
FOR EACH ROW
BEGIN
...
END;
```

Let's examine the syntax above in more detail.

- You put the trigger name after the CREATE TRIGGER statement. The trigger name should follow the naming convention [trigger time]_[table name]_[trigger event], for example before_employees_update.
- Trigger activation time can be BEFORE or AFTER. You must specify the activation time when you define a trigger. You use the BEFORE keyword if you want to process action prior to the change is made on the table and AFTER if you need to process action after the change is made.
- The trigger event can be INSERT, UPDATE or DELETE. This event causes the trigger to be invoked. A trigger only can be invoked by one event. To define a trigger that is invoked by multiple events, you have to define multiple triggers, one for each event.
- A trigger must be associated with a specific table. Without a table trigger would not exist therefore you have to specify the table name after the ON keyword.
- You place the SQL statements between BEGIN and END block. This is where you define the logic for the trigger

Example

```
SQL> Create table account1(acct_num int,amount int)
SQL> insert into account1 values(1,150)
SQL> select * from account1
```

Acc_num	amount
1	150

Trigger for update

```
DELIMITER $$
USE `sample1`$$
CREATE DEFINER = CURRENT_USER TRIGGER `sample1`.`new_table_BEFORE_UPDATE`
BEFORE UPDATE ON `account1` FOR EACH ROW
BEGIN
IF NEW.amount < 0 THEN
SET NEW.amount = 0;
ELSEIF NEW.amount > 100 THEN
SET NEW.amount = 100;
END IF;
END$$
DELIMITER ;
```

```
SQL> update account1 set amount=2000 where acct_num=1
SQL> select * from account1
```

Acc_num	amount
1	100

Trigger for insert

```
DELIMITER $$
USE `sample1`$$
CREATE DEFINER = CURRENT_USER TRIGGER `sample1`.`bank_BEFORE_INSERT` BEFORE
INSERT ON `account1` FOR EACH ROW
BEGIN
IF NEW.amount < 0 THEN
SET NEW.amount = 0;
ELSEIF NEW.amount > 100 THEN
SET NEW.amount = 100;
END IF;
END
$$
DELIMITER ;
```

```
SQL> insert into account1 values(2,-100)
SQL>  select * from account1
```

Acc_num	amount
1	100
2	0

Trigger for Delete

```
DELIMITER $$
USE `sample1`$$
CREATE DEFINER = CURRENT_USER TRIGGER `sample1`.`bank_BEFORE_DELETE` BEFORE
DELETE ON `account1` FOR EACH ROW
BEGIN
delete from account2 where acct_num=2;
END
$$
DELIMITER ;
```

```
SQL> select * from account1

Acc_num     amount
1           100
2           0

SQL> select * from account2
```

Result

Thus the programs for insert, delete, and update operations in a database table using triggers is created and executed successfully.

Ex.No.:09
Date :

View and Index

Aim:

To create and execute the View and Index for the large database and tables.

VIEWS:

SQL includes the ability to create stored queries that they can then be used as a basis for other queries. These stored queries are also called views. A view is simply a derived table that is built upon the base tables of the database. Base tables are the original database tables that actually contain data. Views do not actually store data they are temporary tables. To define a view, we must give the view a name and state the query that computes the view.

Syntax:

Create view v-name as <query expression>

Where query expression is any legal query expression and view name is represented by v-name.(can give any name for view)

SQL> select * from classa;

SID	SNAME	SDEPT	TOTAL
1	aarthi	IT	450
2	ezhil	ECE	590
3	sakthi	IT	900
4	vino	ECE	600
7	viji	IT	900
6	sumathi	ECE	890

SQL> select * from classb;

ID	NAME	GRADE
1	aarthi	b
2	ezhil	b
6	sumathi	a
7	viji	a

CREATING A VIEW:

The first step in creating a view is to define the defining query, which is the query on which the view is based. While it is not required that the defining query be written before creating a view, it is generally a good idea. Any errors in the query can be caught and corrected before the view is created.

SQL> create view V as select classa.sid, classa.sname, classa.sdept, classb.grade from classa, classb where classa.sid=classb.id order by classa.sid;

View created.;

SQL> select * from V;

SID	SNAME	SDEPT	GRADE
1	aarthi	IT	B
2	ezhil	ECE	B

RENAMING COLUMNS IN A VIEW:

Another useful feature available when creating a view is that columns can be renamed in the CREATE VIEW statement. The new column names only apply to the views, the column names in the base tables do not change.

```
SQL> create view classxy(id,dept) as select sid,sdept from classa;
View created.

SQL> select * from classxy;
ID      DEPT
1       IT
2       ECE
3       IT
4       ECE
7       IT
6       ECE
```

Index

An index is a schema object. It is used by the server to speed up the retrieval of rows by using a pointer. It can reduce disk I/O(input/output) by using a rapid path access method to locate data quickly. An index helps to speed up select queries and where clauses, but it slows down data input, with the update and the insert statements. Indexes can be created or dropped with no effect on the data. In this article, we will see how to create, delete, and uses the INDEX in the database.

CREATE INDEX

Syntax

```
CREATE INDEX index_name ON table_name (column1, column2, ...);
```

Example

```
SQL> select * from cus;
```

Cid	cname	address	salary	post
1	aa	77,anna salai,arcot	10500	clerk
3	bb	01,anna salai,chennai	15500	staff
2	cc	25,rajaji nagar,banglore	15500	staff
4	dd	02,mettu street,kochin	10500	secretary
5	ee	21,north street,mumbai	15500	manager

```
SQL>create index cusind on cus(cid,cname);
Index created

SQL> show index from cus;
```

Unique Indexes:

Unique indexes are used for the maintenance of the integrity of the data present in the table as well as for fast performance, it does not allow multiple values to enter into the table.

Syntax:

CREATE UNIQUE INDEX index ON TABLE column;

Delete index

Remove an index from the data dictionary by using the DROP INDEX command.

Syntax:

DROP INDEX index;

Result :

Thus the view and index for a large database and table is created and executed successfully.

Ex. No. 10
Date :

XML database and validate it using XML schema

Aim

Creating XML database and validate it using XML schema.

Procedure

XML

- Xml (eXtensible Markup Language) is a mark up language.
- XML is designed to store and transport data.
- Xml was released in late 90's. it was created to provide an easy to use and store self describing data.
- XML became a W3C Recommendation on February 10, 1998.
- XML is not a replacement for HTML.
- XML is designed to be self-descriptive.
- XML is designed to carry data, not to display data.
- XML tags are not predefined. You must define your own tags.
- XML is platform independent and language independent.

XML Schema

XML Schema is commonly known as XML Schema Definition (XSD). It is used to describe and validate the structure and the content of XML data. XML schema defines the elements, attributes and data types. Schema element supports Namespaces. It is similar to a database schema that describes the data in a database.

How to validate XML against XSD in java:

Java XML Validation API can be used to validate XML against an XSD. javax.xml.validation.Validator class is used in this program to validate xml file against xsd file.Here are the sample XSD and XML files used.

Employee.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.journaldev.com/Employee"
xmlns:empns="http://www.journaldev.com/Employee" elementFormDefault="qualified">
<element name="empRequest" type="empns:empRequest"></element>
<element name="empResponse" type="empns:empResponse"></element>
<complexType name="empRequest">
<sequence>
<element name="id" type="int"></element>
</sequence>
</complexType>
```

<complexType name="empResponse">

```
<sequence>
<element name="id" type="int"></element>
<element name="role" type="string"></elemen>
<element name="fullName" type="string"></element>
</sequence>
</complexType>
</schema>
```

Notice that above XSD contains two root element and namespace also, I have created two sample XML file from XSD.

Employee Request.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<empns:empRequest xmlns:empns="http://www.journaldev.com/Employee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.journaldev.com/Employee Employee.xsd ">
```

EmployeeResponse.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<empns:empResponse xmlns:empns="http://www.journaldev.com/Employee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.journaldev.com/Employee Employee.xsd ">
<empns:id>1</empns:id>
<empns:role>Developer</empns:role>
<empns:fullName>Pankaj Kumar</empns:fullName>
</empns:empResponse>
```

Here is another XML file that doesn't confirms to the Employee.xsd

employee.xml

```
<?xml version="1.0"?>
<Employee>
<name>Pankaj</name>
<age>29</age>
<role>Java Developer</role>
<gender>Male</gender>
</Employee>
```

Here is the program that is used to validate all three XML files against the XSD. The validateXMLSchema method takes XSD and XML file as argument and return true if validation is successful or else returns false.

XMLValidation.java

```
package com.journaldev.xml;
import java.io.File;
import java.io.IOException;
import javax.xml.XMLConstants;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Validator;
import org.xml.sax.SAXException;
public class XMLValidation {
public static void main(String[] args) {
System.out.println("EmployeeRequest.xml validates against Employee.xsd?
"+validateXMLSchema("Employee.xsd", "EmployeeRequest.xml"));
System.out.println("EmployeeResponse.xml validates against Employee.xsd?
"+validateXMLSchema("Employee.xsd", "EmployeeResponse.xml"));
System.out.println("employee.xml validates against Employee.xsd?
"+validateXMLSchema("Employee.xsd", "employee.xml"));
}
public static boolean validateXMLSchema(String xsdPath, String xmlPath){
try {
SchemaFactory factory =
SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = factory.newSchema(new File(xsdPath));
Validator validator = schema.newValidator();
validator.validate(new StreamSource(new File(xmlPath)));
} catch (IOException | SAXException e) {
System.out.println("Exception: "+e.getMessage());
return false;
}
return true;
}
}
```

Output of the above program is:

```
EmployeeRequest.xml validates against Employee.xsd? true
EmployeeResponse.xml validates against Employee.xsd? true
Exception: cvc-elt.1: Cannot find the declaration of element 'Employee'.
employee.xml validates against Employee.xsd? false
```

RESULT:

Thus the XML database created and validates it using XML schema

Ex. No.: 11
Date :

NoSQL Database Tools

Aim:

To create the document, columns and graphs based on data by using NoSQL tools.

Procedure :

Document database

A document database (also known as a document-oriented database or a document store) is a database that stores information in documents.

A document is a record in a document database. A document typically stores information about one object and any of its related metadata.

Documents store data in field-value pairs. The values can be a variety of types and structures, including strings, numbers, dates, arrays, or objects. Documents can be stored in formats like JSON, BSON, and XML.

Below is a JSON document that stores information about a user named Tom.

```
{
    "_id": 1,
    "first_name": "Tom",
    "email": "tom@example.com",
    "cell": "765-555-5555",
    "likes": [
      "fashion",
      "spas",
      "shopping"
    ],
    "businesses": [
      {
        "name": "Entertainment 1080",
        "partner": "Jean",
        "status": "Bankrupt",
        "date_founded": {
          "$date": "2012-05-19T04:00:00Z"
        }
      },
      {
        "name": "Swag for Tweens",
        "date_founded": {
          "$date": "2012-11-01T04:00:00Z"
        }
      }
    ]
```

```
}
```

Collections
A collection is a group of documents. Collections typically store documents that have similar contents.

Continuing with the example above, the document with information about Tom could be stored in a collection named users. More documents could be added to the users collection in order to store information about other users. For example, the document below that stores information about Donna could be added to the users collection.

```
{
    "_id": 2,
    "first_name": "Donna",
    "email": "donna@example.com",
    "spouse": "Joe",
    "likes": [ "spas",
      "shopping",
      "live tweeting"
    ],
    "businesses": [
      {
        "name": "Castle Realty",
        "status": "Thriving",
        "date_founded": {
          "$date": "2013-11-21T04:00:00Z"
        }
      }
    ]
 }
```

Columnar Data Model of NoSQL :
In Columnar Data Model instead of organizing information into rows, it does in columns. This makes them function the same way that tables work in relational databases. This type of data model is much more flexible obviously because it is a type of NoSQL database.
The below example will help in understanding the Columnar data model:

Row-Oriented Table:

S.No.	Name	Course	Branch	ID
01.	Tanmay	B-Tech	Computer	2
02.	Abhishek	B-Tech	Electronics	5

03.	Samriddha	B-Tech	IT	7

S.No.	Name	Course	Branch	ID
04.	Aditi	B-Tech	E & TC	8

Column – Oriented Table:

S.No.	Name	ID
01.	Tanmay	2
02.	Abhishek	5
03.	Samriddha	7
04.	Aditi	8

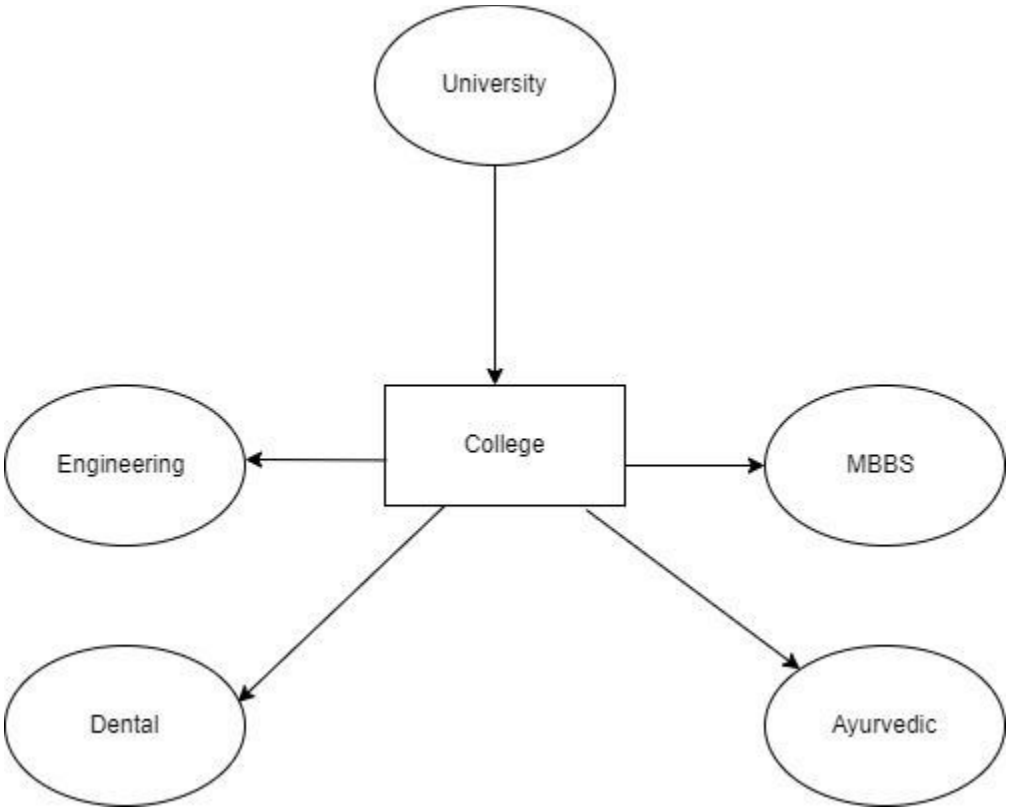
Graph Database on NoSQL

Graph Based Data Model in NoSQL is a type of Data Model which tries to focus on building the relationship between data elements. As the name suggests Graph-Based Data Model, each element here is stored as a node, and the association between these elements is often known as Links. Association is stored directly as these are the first-class elements of the data model. These data models give us a conceptual view of the data.

These are the data models which are based on topographical network structure. Obviously, in graph theory, we have terms like Nodes, edges, and properties, let’s see what it means here in the Graph-Based data model.

Nodes: These are the instances of data that represent objects which is to be tracked.
Edges: As we already know edges represent relationships between nodes.
Properties: It represents information associated with nodes.

The below image represents Nodes with properties from relationships represented by edges.



Result :

Thus we studied the various NoSQL database tools to create document, column and graph successfully.

Ex.No.12
Date :

Simple GUI Application
(Displaying Student Mark List)

Aim:

Write a program in Java to create Displaying student mark listusing JSP and Databases (three tire architecture).

Definition, usage and procedure

Three tier architecture is a very common architecture. A three tier architecture is typically split into a presentation or GUI tier, an application logic tier, and a data tier.

Presentation tier encapsulates the presentation logic required to serve clients. A JSP in the presentation tier intercepts client requests, manages logons, sessions, accesses the business services, and finally constructs a response, which gets delivered to client.

Business tier provides the business services. This tier contains the business logic and the business data. All the business logic is centralized into this tier as opposed to 2-tier systems where the business logic is scattered between the front end and the backend. The benefit of having a centralized business tier is that same business logic can support different types of clients like browser, WAP (Wireless Application Protocol) client. In our exercise we will use servlet as business tier.

Data Tier

Data tier is used by the databases

JSP

Java Server Pages (JSP) is a server-side programming technology that enables the creation of dynamic, platform-independent method for building Web-based applications. JSP have access to the entire family of Java APIs, including the JDBC API to access enterprise databases

Servlet

A **servlet** is a small Java program that runs within a Web server. **Servlets** receive and respond to requests from Web clients, usually across HTTP, the HyperText Transfer Protocol

Client:

Step1: In index.html on the client side declare the contents that you like to transfer to the server using html form and input type tags.
Step2: create a submit button and close all the included tags.

Servlet:

Step 1: Import all necessary packages
Step 2: Define a class that extends servlet
Step 3: In the doPost() method, do the following:

i) Set the content type of the response to "text/html"
ii) connect with the database which has the student marklist
iii) query the data to the database

Step 4: Display the student marklist

First Create database as **db8** in that create table as **mark** with the following field

create table mark(rno varchar(20),name1 varchar(20),m1 varchar(20),m2 varchar(20),m3 varchar(20),m4 varchar(20),m5 varchar(20),m6 varchar(20))

insert into mark values('100','mohammed','90','90','90','90','90','90')

select * from mark

index.html

```
<head>
<title>Three Tier Application</title>
<style type="text/css">
body{color:blue;font-family:courier;text-align:center}
</style>
</head>
<body>
<h2>EXAMINATION RESULT</h2><hr/>
<form name="f1" method="GET" action="marklist.jsp">
Enter Your Reg.No:
<input type="text" name="rno"/><br/><br/>
       
    
<input type="submit" value="SUBMIT"/>
</form>
</body>
</html>
```

<html>

Marklist.jsp

```
<%@page import="java.util.Properties"%>
<%@page contentType="text/html" language="java" import="java.sql.*"%>
<html>
<head>
<title>Three Tier Application</title>
<style type="text/css">
body{color:blue;font-family:courier;text-align:center}
</style>
</head>
<body>
<h2>EXAMINATION RESULT</h2><hr/>
<%
String str=request.getParameter("rno");
Class.forName("org.apache.derby.jdbc.ClientDriver");
Properties p=new Properties();
```

```
p.put("user","root");
```

```
p.put("password","root");
Connection con=DriverManager.getConnection("jdbc:derby://localhost:1527/db8",p);
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery( " Select * FROM mark WHERE rno ="'+str+"'");
while(rs.next())
{
%>
Register No:<%=rs.getString(1)%><br/>
Name:<%=rs.getString(2)%><br/>
<table border="1">
<th>SUBJECT</th><th>Mark</th>
<tr><td>DBMS</td><td><%=rs.getString(3)%></td></tr>
<tr><td>TOC</td><td><%=rs.getString(4)%></td></tr>
<tr><td>AI</td><td><%=rs.getString(5)%></td></tr>
<tr><td>OS</td><td><%=rs.getString(6)%></td></tr>
<tr><td>Algorithms</td><td><%=rs.getString(7)%></td></tr>
<tr><td>EVS</td><td><%=rs.getString(8)%></td></tr>
</table>
<%
}
%>
<br/>
<a href="index.jsp">Back</a>
</body>
</html>
```

Output

Result :

Thus the program is created for displaying examination result and successfully verified

Ex. No. 13
Date :

Case Study of Cop Friendly App – Eseva

Aim

To case Study of Cop Friendly APP –Eseva

About Case study

The Case Study is all about Smart Cop, a mobile application or tool developed by Sapio Analytics along with Dinosys Infotech that places a strong emphasis on combining existing interdisciplinary information, improving analysis techniques, and developing more efficient police strategies. For example, the Data-Driven Approach to Crime Scene Management is a technology that specializes in handling evidence collected at a crime scene, and its notification feature aids in pulling data from various media. Escape routes and the location of the incident could also be tracked. Also, describes how the police can effectively manage their limited resources with this new data-driven and AI-based policing

Purpose of this Case Study

The purpose of the case study is to demonstrate how using the SMART COP tool developed by Sapio Analytics along with Dinosys Infotech when used on a daily basis in policing can help police agencies to figure out what works in crime reduction and crime prevention initiatives, as well as lead to much more effective decisions, faster actions, which can lead to better policing, higher impact on citizens, and improved citizen satisfaction. It's to demonstrate the value of such efficacious, efficient, and cost effective law enforcement strategies and tactics based on data and analytics, especially when it's aided by Sapio Analytics' exclusive systems.

Background of the Product

Sapio Analytics along with Dinosys Infotech is providing training consultations and analytical solutions backed by Artificial Intelligence to the enforcement bodies in matters related to cyber-crime, suspect profiling, predictive policing, and so on. The Smart Cop tool developed by Sapio Analytics along with Dinosys is an essential guarantee to modernize the police to manage local security. It will also emphasize the new exigencies of the police management system. It can likewise be utilized for mappingcrime which can monitor high-crime locations. With this, police can monitor very closely and have a real-time pulse on criminal activities. The mounted Smart Cop App will enhance the balance, security, sustenance, and scalability of the existing department. The application facilitates an integrated policing management system along with an information processing platform that will empower front-line Police Officers who are on a beat towards achieving Global Policing Standards

Problem Statement

India is known as the world's largest democratic country, little is known about how it polices such a vast, complex, and unpredictable country. As a result, police officers encounter challenges and barriers in carrying out their duties on a daily basis. Some of the problems faced are

The police leadership has not placed a high priority on using technology to deliver services to citizens.

- Investigations are being delayed due to a lack of collaboration between internal divisions.
- The training standards are quite inadequate and do not account for the use of new technology.

- There is a significant disparity between the rate at which crimes are committed and the rate at which FIRs are filed.
- The workload is one of the key causes of police inefficiency once again.

Solution to the Problem

The fundamental function of police forces is to uphold and investigate crimes, safeguard the safety of citizens, and enforce laws. In a large and populous country like India, police forces must be well equipped to fulfill their duties effectively. As a result, the police force must adjust to changing conditions. Police modernization has been needed for data protection, counterterrorism/insurgency, and reliance on technology for policing. This necessitates more investment in technological advancement and modernization. So, to solve the problem in a modern way Sapio Analytics Came up with Smart Cop which aims to train the law enforcement agencies on emerging technology, mitigating cybercrimes, enhancing their skill set as well as capacity building so that they are combatready in a real-time basis. The Smart Cop is an essential guarantee to modernize the police to manage local security. It will also emphasize the new exigencies of the police management system.

Benefits of the App

The main aim of Smart Cop is to digitize the whole functioning of beats. In the due process, we facilitated the beat police with one application and have integrated the IT applications and databases which are in line with the beat system, and converged them on to Smart Cop

- Effectively utilizing Information and Communication Technology (ICT) traversing from E-Governance to M-Governance
- Empowering and delegating the frontline Beat Police Officers for demonstrating quick & smart decision making
- Delivering Services 'Anytime & Anywhere' for faster response to the Citizens Seamless integration of different application functionalities through Single-Sign-On services
- Efficient identification and tracking of suspects, quicker resolution of cases, and increased rate of convictions for the offenders
- Proactively preventing crimes through real-time intelligence inputs and analysis relating to crimes and criminals
- Encouraging transparency and accountability in every police personnel

Conclusive Summary

The case study shows how as the rate of crime rises; the utilization of existing Artificial Intelligence algorithms is proving to be extremely beneficial. The tool developed by Sapio Analytics along with Dinosys Infotech- SMART COP sets a promising example. To a considerable extent, Smart Cop aids in the prediction of crime as well as the criminal. Artificial intelligence has the potential to become a permanent element of the criminal justice system. Technological reforms are required to accomplish the vision of SMART policing, it's important to train the police for new challenges, and strengthen their investigative and emergency response capabilities. This will eventually increase public confidence in the police force's effectiveness and its ability to serve efficiently. The police force must be eager to bring a change and adopt new-age technologies and systems into the realm of law enforcement for it to be more proactive than reactive.

INVENTORY MANAGEMENT SYSTEM

PROBLEM STATEMENT:

INVENTORY MANAGEMENT SYSTEM is a real time application used in the merchant's day to day system. This is a database to store the transaction that takes places between the Manufacturer, Dealer and the Shop Keeper that includes stock inward and stock outward with reference to the dealer. Here we assume our self as the Dealer and proceed with the transaction as follows:

The Manufacturer is the producer of the items and it contains the necessary information of the item such as price per item, Date of manufacture, best before use, Number of Item available and their Company Address.

The Dealer is the secondary source of an Item and he purchases Item from the manufacturer by requesting the required Item with its corresponding Company Name and the Number of Items required. The Dealer is only responsible for distribution of the Item to the Retailers in the Town or City.

The Shop Keeper or Retailer is the one who is prime source for selling items in the market. The customers get Item from the Shop Keeper and not directly from the Manufacturer or the Dealer.

The Stock is the database used in our System which records all transactions that takes place between the Manufacturer and the Dealer and the Dealer and the Retailer.

CODING:

FORM1

```
Dim db As Database
Dim rs As Recordset

Private Sub Command1_Click()
Form3.Show
End Sub

Private Sub Command2_Click()
Form4.Show
End Sub

Private Sub Command3_Click()
Form5.Show
End Sub

Private Sub Command4_Click()
End
End Sub

Private Sub Form_Load()
Set db = OpenDatabase("D:\prj789\invent\INVENTORY.MDB")
Set rs = db.OpenRecordset("SYSTEM")

End Sub
```

FORM2

```
Dim db As Database
Dim rs As Recordset
Dim i As Integer

Private Sub Command1_Click()
Form1.Show
 End Sub

Private Sub Command2_Click()
rs.MoveFirst
For i = 1 To rs.RecordCount
    If rs(0) = Text1.Text Then
    rs.Edit

   If Text7.Text = "" Then
   MsgBox "enter the no of items ordered"
   Else
    rs(6) = Text7.Text
    rs(7) = rs(5) * rs(6)
    rs(4) = rs(4) + Val(Text7.Text)
    Text8.Text = rs(7)
    Text5.Text = rs(4)
    Text4.Text = rs(3)
    rs.Update
    GoTo l1
     End If
     End If
 rs.MoveNext
 Next i
l1: End Sub

Private Sub Command3_Click()
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
Text4.Text = ""
Text5.Text = ""
Text6.Text = ""
Text7.Text = ""
Text8.Text = ""
End Sub

Private Sub Command4_Click()
```

```
rs.AddNew
```

```
rs(0) = Text1.Text
rs(1) = Text2.Text
rs(2) = Text3.Text
rs(3) = Text4.Text
rs(4) = Text5.Text
rs(5) = Text6.Text


rs(6) = Text7.Text
rs(7) = Text8.Text
rs.Update
End Sub

Private Sub Form_Load()
Set db = OpenDatabase("D:\prj789\invent\INVENTORY.MDB")
Set rs = db.OpenRecordset("SYSTEM")
End Sub

Private Sub List1_Click()
Text1.Text = List1.Text
rs.MoveFirst
For i = 1 To rs.RecordCount
If rs(0) = Text1.Text Then
   Text2.Text = rs(1)
   Text3.Text = rs(2)
   Text4.Text = rs(3)
   Text5.Text = rs(4)
   Text6.Text = rs(5)
   Text7.Text = ""
   Text8.Text = ""
End If
rs.MoveNext
Next i
End Sub
```

FORM3

```
Dim db As Database
Dim rs As Recordset
Dim i As Integer

Private Sub Command1_Click()
rs.MoveFirst
For i = 1 To rs.RecordCount
   If rs(0) = Text1.Text Then
```

```
            rs.Edit
            If Text4.Text = "" Then
            MsgBox "Enter the no of items needed"
            Else
            rs(6) = Text4.Text
            If rs(6) <= rs(4) Then
            rs(7) = rs(5) * rs(6)
            rs(4) = rs(4) - Val(Text4.Text)
            Text2.Text = rs(4)
            Text5.Text = rs(7)
            Else
            MsgBox " ITEM NOT SUFFICIENT"
            End If
            rs.Update
            GoTo l1



    End If
    End If
 rs.MoveNext
 Next i
l1: End Sub

Private Sub Command2_Click()
Form1.Show
End Sub

Private Sub Command3_Click()
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
Text4.Text = ""
Text5.Text = ""
End Sub

Private Sub Form_Load()
Set db = OpenDatabase("D:\prj789\invent\INVENTORY.MDB")
Set rs = db.OpenRecordset("SYSTEM")
End Sub

Private Sub List2_Click()
Text1.Text = List2.Text
rs.MoveFirst
For i = 1 To rs.RecordCount
```

```
If rs(0) = Text1.Text Then
```

```
    Text2.Text = rs(4)
    Text3.Text = rs(5)
    Text4.Text = ""
    Text5.Text = ""
End If
rs.MoveNext
Next i
End Sub
```

FORM4

```
Dim db As Database
Dim rs As Recordset
Dim r, i As Integer

Private Sub Command1_Click()
Form1.Show
End Sub

Private Sub Form_Load()
Set db = OpenDatabase("D:\prj789\invent\INVENTORY.MDB")
Set rs = db.OpenRecordset("SYSTEM")

MSFlexGrid1.FixedRows = 0
MSFlexGrid1.FixedCols = 0


r = 0
MSFlexGrid1.ColWidth(0) = 2000
MSFlexGrid1.ColWidth(1) = 2000
MSFlexGrid1.ColWidth(2) = 2000
MSFlexGrid1.ColWidth(3) = 1700
MSFlexGrid1.ColWidth(4) = 1750
MSFlexGrid1.ColWidth(5) = 1650

 'MSFlexGrid1.ForeColor = "GREEN"
 MSFlexGrid1.TextMatrix(0, 0) = "COMPANY NAME"
 MSFlexGrid1.TextMatrix(0, 1) = "COMPANY ADDRESS"
 MSFlexGrid1.TextMatrix(0, 2) = "CONTACT NUMBER"
 MSFlexGrid1.TextMatrix(0, 3) = "DATE OF ORDER"
 MSFlexGrid1.TextMatrix(0, 4) = "ITEMS AVAILABLE"
 MSFlexGrid1.TextMatrix(0, 5) = "PRICE/ITEM"
 rs.MoveFirst
 r = 1
Do Until rs.EOF
```

```
MSFlexGrid1.FixedRows = r
MSFlexGrid1.FixedCols = 0
MSFlexGrid1.Text = rs(0)
MSFlexGrid1.FixedRows = r
MSFlexGrid1.FixedCols = 1
MSFlexGrid1.Text = rs(1)
MSFlexGrid1.FixedRows = r
MSFlexGrid1.FixedCols = 2
MSFlexGrid1.Text = rs(2)
MSFlexGrid1.FixedRows = r
MSFlexGrid1.FixedCols = 3
MSFlexGrid1.Text = rs(3)
MSFlexGrid1.FixedRows = r
MSFlexGrid1.FixedCols = 4
MSFlexGrid1.Text = rs(4)
MSFlexGrid1.FixedRows = r
MSFlexGrid1.FixedCols = 5
MSFlexGrid1.Text = rs(5)
MSFlexGrid1.FixedRows = r
MSFlexGrid1.FixedCols = 6
'MSFlexGrid1.Text = rs(6)
'MSFlexGrid1.FixedRows = r
'MSFlexGrid1.FixedCols = 7
'MSFlexGrid1.Text = rs(7)
r = r + 1
rs.MoveNext
Loop
End Sub
```

FORMS

FORM1 : MAIN MENU

FORM2 : PURCHASE DETAILS

FORM3 : SALES DETAILS

FORM4 :STOCK DETAILS