Practical No: 1

Title: ER Modeling and Normalization:

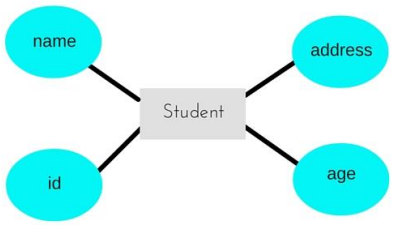
Decide a case study related to real time application in group of 2-3 students and formulate a problem statement for application to be developed. Propose a Conceptual Design using ER features using tools like ERD plus, ER Win etc. (Identifying entities, relationships between entities, attributes, keys, cardinalities, generalization, specialization etc.) Convert the ER diagram into relational tables and normalize Relational data model.

Requirements:

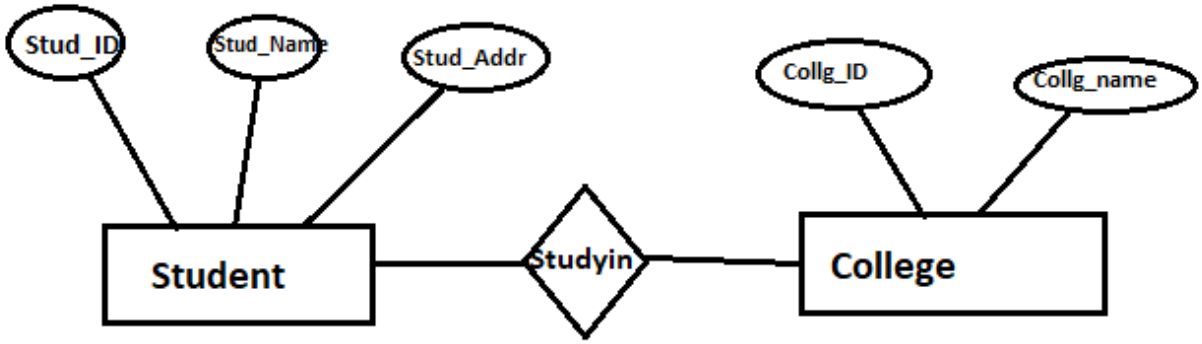
ERD plus, ER Win

Theory:

- The ER data mode was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical structure of a database.
- The ER model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. Because of this usefulness, many database-design tools draw on concepts from the ER model.
- The ER data model employs three basic concepts:
- entity sets,
- relationship sets,
- attributes.
- The ER model also has an associated diagrammatic representation, the ER diagram, which can express the overall logical structure of a database graphically.
- An Entity Relationship(E-R) model is systematic way of describing and defining a conceptual database design process.
- An E-R model is defined as, " a conceptual data model that views the real-world as entities and relationships".

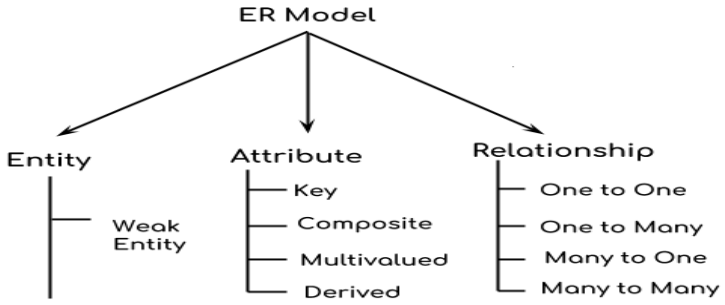


- An Entity–relationship model (ER model) describes the structure of a database with the help of a diagram, which is known as Entity Relationship Diagram (ER Diagram).



Symbols in E-R Diagram

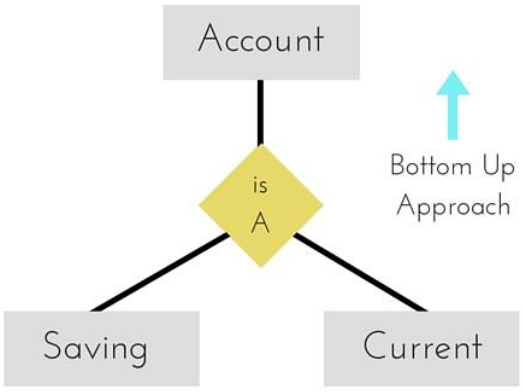
- Rectangle: Entity/Entity sets(strong)
- Double Rectangle: Entity sets(weak)
- Ellipses: Attributes
- Diamonds: Relationship Set
- Lines: They link attributes to Entity Sets and Entity sets to Relationship Set
- Double Ellipses: Multivalued Attributes
- Dashed Ellipses: Derived Attributes
- Double Lines: Total participation of an entity in a relationship set



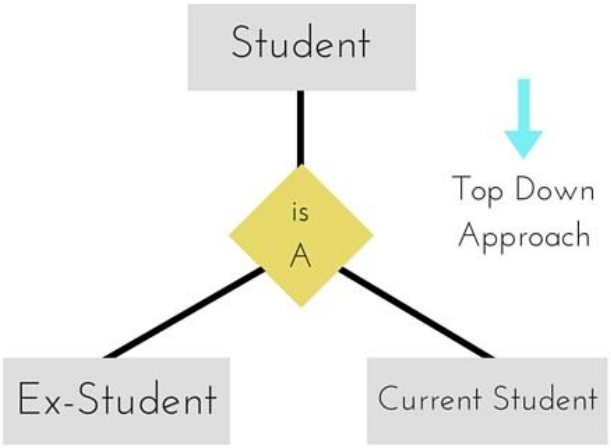
Components of ER Diagram

- The E-R model that is supported with the additional semantic concepts called the Enhanced Entity- relationship model (EER) model. they were:
- Generalization
- Specialization
- Aggregation
- Subclass and Superclass
- Category or Union

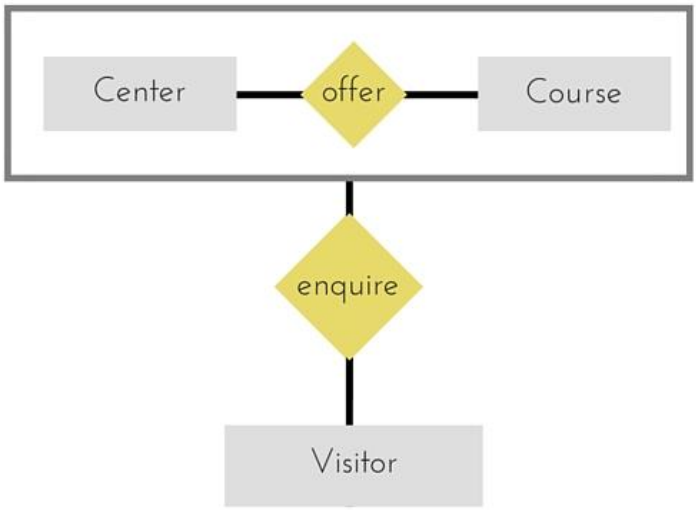
- **Generalization** is a bottom-up approach in which two lower level entities combine to form a higher level entity. In generalization, the higher level entity can also combine with other lower level entities to make further higher level entity.



Specialization is opposite to Generalization. It is a top-down approach in which one higher level entity can be broken down into two lower level entity. In specialization, a higher level entity may not have any lower-level entity sets, it's possible.



- **Aggregation** is a process when relation between two entities is treated as a single entity.



Conversion of ER diagram to Table

The database can be represented using the notations, and these notations can be reduced to a collection of tables.

In the database, every entity set or relationship set can be represented in tabular form

- **Entity type becomes a table.**

In the given ER diagram, LECTURE, STUDENT, SUBJECT and COURSE forms individual tables.

- **All single-valued attribute becomes a column for the table.**

In the STUDENT entity, STUDENT_NAME and STUDENT_ID form the column of STUDENT table. Similarly, COURSE_NAME and COURSE_ID form the column of COURSE table and so on.

- **A key attribute of the entity type represented by the primary key.**

In the given ER diagram, COURSE_ID, STUDENT_ID, SUBJECT_ID, and LECTURE_ID are the key attribute of the entity.

- **The multivalued attribute is represented by a separate table.**

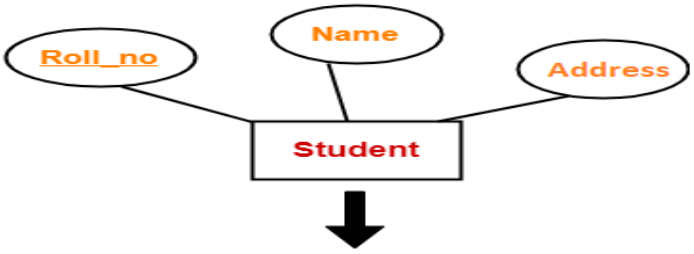
In the student table, a hobby is a multivalued attribute. So it is not possible to represent multiple values in a single column of STUDENT table. Hence we create a table STUD_HOBBY with column name STUDENT_ID and HOBBY. Using both the column, we create a composite key.

- **Composite attribute represented by components.**

In the given ER diagram, student address is a composite attribute. It contains CITY, PIN, DOOR#, STREET, and STATE. In the STUDENT table, these attributes can merge as an individual column.

- **Derived attributes are not considered in the table.**

In the STUDENT table, Age is the derived attribute. It can be calculated at any point of time by calculating the difference between current date and Date of Birth.
Using these rules, you can convert the ER diagram to tables and columns and assign the mapping between the tables. Table structure for the given ER diagram is as below:



Roll_no	Name	Address

Normalization

- Normalization is the process of organizing the data in the database.
- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.
- Normalization divides the larger table into the smaller table and links them using relationship.
- The normal form is used to reduce redundancy from the database table.

1NF	A relation is in 1NF if it contains an atomic value.
2NF	A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.
3NF	A relation will be in 3NF if it is in 2NF and no transition dependency exists.
4NF	A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
5NF	A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.

Conclusion :

Hence we conclude that we studying ER diagram and features using tools and how to normalize the table

Questions:

1. Draw ER diagram for college ERP system.?
2. Design an E-R diagram with EER features which will model all the entities and relationships among them for the Airline Reservation System Database.

Assignment No: 2

Title:

a. Design and Develop SQL DDL statements which demonstrate the use of SQL objects such as Table, View, Index, Sequence, Synonym, different constraints etc.

b. Write at least 10 SQL queries on the suitable database application using SQL DML statements which demonstrate the use of concepts like Insert, Select, Update, Delete with operators, functions, and set operator.

Requirements:

1. Computer System with Linux/Open Source Operating System.
2. Mysql Server

Theory:

Data Definition Language (DDL): Data Definition Language (DDL) or Schema Definition Language, statements are used to define the database structure or schema.

1. CREATE - to create objects in the database
2. ALTER - alters the structure of the database
3. DROP - delete objects from the database
4. TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
5. COMMENT - add comments to the data dictionary
6. RENAME- renames an object.

Create Table Statement:

Creating a basic table involves naming the table and defining its columns and each column's data type. The SQL **CREATE TABLE** statement is used to create a new table.

Syntax:

Basic syntax of CREATE TABLE statement is as follows:

```
CREATE TABLE table_name(
column1 datatype,
column2 datatype,
column3 datatype,
.....
columnN datatype,
PRIMARY KEY( one or more columns )
);
```

CREATE TABLE is the keyword telling the database system what you want to do. In this case, you want to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement. Then in brackets comes the list defining each column in the table and what sort of data type it is. The syntax becomes clearer with an example below. A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement.

You can verify if your table has been created successfully by looking at the message displayed by the SQL server, otherwise you can use **DESC** command as follows:

DESC table_name;

DROP TABLE:

The SQL **DROP TABLE** statement is used to remove a table definition and all data, indexes, triggers, constraints, and permission specifications for that table.

NOTE: You have to be careful while using this command because once a table is deleted then all the information available in the table would also be lost forever.

Syntax:

Basic syntax of DROP TABLE statement is as follows:

DROP TABLE table_name;

ALTER TABLE

SQL **ALTER TABLE** command is used to add, delete or modify columns in an existing table. You would also use ALTER TABLE command to add and drop various constraints on an existing table.

Syntax:

The basic syntax of **ALTER TABLE** to add a new column in an existing table is as follows:

ALTER TABLE table_name ADD column_name datatype;

The basic syntax of ALTER TABLE to **DROP COLUMN** in an existing table is as follows:

ALTER TABLE table_name DROP COLUMN column_name;

The basic syntax of ALTER TABLE to change the **DATA TYPE** of a column in a table is as follows:

ALTER TABLE table_name MODIFY COLUMN column_name datatype;

The basic syntax of ALTER TABLE to add a **NOT NULL** constraint to a column in a table is as follows:

ALTER TABLE table_name MODIFY column_name datatype NOT NULL;

The basic syntax of ALTER TABLE to **ADD UNIQUE CONSTRAINT** to a table is as follows:

ALTER TABLE table_name
ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...);

The basic syntax of ALTER TABLE to **ADD CHECK CONSTRAINT** to a table is as follows:
ALTER TABLE table_name
ADD CONSTRAINT MyUniqueConstraint CHECK (CONDITION);

The basic syntax of ALTER TABLE to **ADD PRIMARY KEY** constraint to a table is as follows:
ALTER TABLE table_name
ADD CONSTRAINT MyPrimaryKey PRIMARY KEY (column1, column2...);

The basic syntax of ALTER TABLE to **DROP CONSTRAINT** from a table is as follows:
ALTER TABLE table_name
DROP CONSTRAINT MyUniqueConstraint;

If you're using MySQL, the code is as follows:
ALTER TABLE table_name
DROP INDEX MyUniqueConstraint;

The basic syntax of ALTER TABLE to **DROP PRIMARY KEY** constraint from a table is as follows:
ALTER TABLE table_name
DROP CONSTRAINT MyPrimaryKey;

If you're using MySQL, the code is as follows:
ALTER TABLE table_name
DROP PRIMARY KEY;

TRUNCATE TABLE
SQL **TRUNCATE TABLE** command is used to delete complete data from an existing table.
You can also use DROP TABLE command to delete complete table but it would remove complete table structure form the database and you would need to re-create this table once again if you wish you store some data.

The basic syntax of **TRUNCATE TABLE** is as follows:
TRUNCATE TABLE table_name;

VIEW in SQL
A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.
A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depend on the written SQL query to create a view. Views, which are kind of virtual tables, allow users to do the following:

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data such that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

Creating Views:
Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables, or another view.
To create a view, a user must have the appropriate system privilege according to the specific implementation.
The basic CREATE VIEW syntax is as follows:
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
You can include multiple tables in your SELECT statement in very similar way as you use them in normal SQL SELECT query.

The View WITH CHECK OPTION

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition. If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following code block has an example of creating same view CUSTOMERS_VIEW with the WITH CHECK OPTION.

```
CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS
WHERE age IS NOT NULL
WITH CHECK OPTION;
```

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

Dropping Views

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple and is given below −
Syntax:DROP VIEW view_name;

SQL Index:
Index are special lookup tables that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book. For example, if you want to reference all pages in a book that discuss a certain topic, you first refer to the index, which lists all topics alphabetically and are then referred to one or more specific page numbers. An index helps speed up SELECT queries and WHERE clauses, but it slows down data input, with UPDATE and INSERT statements.

Indexes can be created or dropped with no effect on the data. Creating an index involves the CREATE INDEX statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in ascending or descending order. Indexes can also be unique, similar to the UNIQUE constraint, in that the index prevents duplicate entries in the column or combination of columns on which there's an index.

The basic syntax of **CREATE INDEX** is as follows:
CREATE INDEX index_name ON table_name;

Single-ColumnIndexes:
A single-column index is one that is created based on only one table column. The basic syntax is as follows:

CREATE INDEX index_name
ON table_name (column_name);

Unique Indexes:
Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows:

CREATE INDEX index_name
on table_name (column_name);

Composite Indexes:
A composite index is an index on two or more columns of a table. The basic syntax is as follows:
CREATE INDEX index_name Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

Implicit Indexes:
Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

The DROP INDEX Command:
An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because performance may be slowed or improved.
The basic syntax is as follows:

DROP INDEX index_name; on table_name (column1, column2);

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

When should indexes be avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided. The following guidelines indicate when the use of an index should be reconsidered:

- Indexes should not be used on small tables.
- Tables that have frequent, large batch update or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

Sequence in SQL

A sequence is a set of integers 1, 2, 3, ... that are generated in order on demand. Sequences are frequently used in databases because many applications require each row in a table to contain a unique value, and sequences provide an easy way to generate them. This chapter describes how to use sequences in MySQL.

Using AUTO_INCREMENT column:

The simplest way in MySQL to use sequences is to define a column as AUTO_INCREMENT and leave rest of the things to MySQL to take care.

Obtain AUTO_INCREMENT Values:

LAST_INSERT_ID( ) is a SQL function, so you can use it from within any client that understands how to issue SQL statements. Otherwise, PERL and PHP scripts provide exclusive functions to retrieve auto-incremented value of last record.

Starting a Sequence at a Particular Value:

By default, MySQL will start sequence from 1 but you can specify any other number as well at the time of table creation. Following is the example where MySQL will start sequence from 100.

```
mysql> CREATE TABLE INSECT
-> (
-> id INT UNSIGNED NOT NULL AUTO_INCREMENT = 100,
-> PRIMARY KEY (id),
-> name VARCHAR(30) NOT NULL, # type of insect
-> date DATE NOT NULL, # date collected
-> origin VARCHAR(30) NOT NULL # where collected
);
```

Alternatively, you can create the table and then set the initial sequence value with ALTER TABLE.

```
mysql> ALTER TABLE t AUTO_INCREMENT = 100;
```

Synonym in SQL

A **synonym** is an alternative name for objects such as tables, views, sequences, stored procedures, and other database objects. You generally use synonyms when you are granting access to an object from another schema and you don't want the users to have to worry about knowing which schema owns the object.

SQL> create synonym emp for employee;

Synonym created.

SQL> select * from emp;

NAME	ROLLNO	BRANCH
ram	4	etc
ramesh	1	comp
geeta	2	IT
shyam	3	mech

1) **MySQL INSERT Query:**

Syntax:

INSERT INTO table_name(field1, field2,...fieldN) VALUES (value1, value2,...valueN);

2) **MySQL UPDATE Query:**

Syntax:

UPDATE table_name SET field1=new-value1, field2=new-value2[WHERE Clause];

3) **MySQL DELETE Query:**

Syntax:

DELETE FROM table_name [WHERE Clause];

4) **MySQL SELECT Query:**

Syntax:

SELECT field1, field2,...fieldN from table WHERE Clause;

5) **MySQL UPDATE Query:**

Syntax:

UPDATE table_name SET field1=new-value1 [WHERE Clause];

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations. Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators
- Logical operators
- Operators used to negate conditions

SQL Arithmetic Operators:

Operator	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder

SQL Comparison Operators:

Operator	Description
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.

SQL Logical Operators:

Operator	Description
ALL	The ALL operator is used to compare a value to all values in another value set.
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
IN	The IN operator is used to compare a value to a list of literal values that have

	been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.**
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
ANY	The ANY operator is used to compare a value to any applicable value in the list according to the condition.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.

Aggregation Functions:

Operator	**Description**
AVG()	Return the average value of the Arguments
BIT_AND()	Return bitwise and
BIT_OR()	Return bitwise or
BIT_XOR()	Return bitwise xor
COUNT(DISTINCT)	Return the count of a number of different values
COUNT()	Return a count of the number of rows returned
GROUP_CONCAT()	Return a concatenated string
MAX()	Return the maximum value
MIN()	Return the minimum value
STD()	Return the population standard derivation
STDDEV_POP()	Return the population standard deviation
STDDEV_SAMP()	Return the sample standard deviation
STDDEV()	Return the population standard derivation
SUM()	Return the sum

Set Operator:

1. **Union**

UNION is used to combine the results of two or more Select statements. However it will eliminate duplicate rows from its result set. In case of union, number of columns and datatype must be same in both the tables.

SQL syntax for Union query:

select * from FirstUNION select * from second;

2. **Union All**

This operation is similar to Union. But it also shows the duplicate rows.

SQL syntax for Union All query:

select * from First UNION ALL select * from second;

3. **Intersect**

ntersect operation is used to combine two SELECT statements, but it only retuns the records which are common from both SELECT statements. In case of Intersect the number of columns and datatype must be same. MySQL does not support INTERSECT operator.

SQL syntax for Intersect query:

select * from First INTERSECT select * from second;

4. **Minus**

Minus operation combines result of two Select statements and return only those result which belongs to first set of result. MySQL does not support INTERSECT operator.

SQL syntax for Minus query:

select * from First MINUS select * from second;

Questions

1. What is the need of view also explain need of index in DBMS?
2. What is the differences between drop and truncate and delete?
3. Consider the relational database
   Supplier (sid, sname, address)
   Parts (pid, pname, color)
   Catalog (sid, pid, cost)
   Write SQL queries for the following requirements: (any 2)
   i) Find name of all parts whose color is green.
   ii) Find names of suppliers who supply some red parts.
   iii) Find names of all parts whose cost is more than Rs. 25
4. Consider relational schema
   Customer (cname, ccity, phone)
   Loan (lno, branch_name, amount)
   Borrower(cname, lno)
   Depositor(cname,accno)

Branch(bname, bcity)
Account(bname, accno, bal)

Write SQL queries for following requirements: (Any two)

i) Find the names of customers whose city name includes ‘bad’.
ii) Find all customers who have an account but no loan in the bank.
iii) Find out average account balance at each branch.

Assignment No: 2

Title: Design at least 10 SQL queries for suitable database application using SQL DML statements: all types of Join, Sub-Query and View.

Requirements:

1. Computer System with Linux/Open Source Operating System.
2. Mysql Server

JOINS: The SQL **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.
Types of Joins
SQL Join Types:
There are different types of joins available in SQL:

- **INNER JOIN:** returns rows when there is a match in both tables.
- **LEFT JOIN:** returns all rows from the left table, even if there are no matches in the right table.
- **RIGHT JOIN:** returns all rows from the right table, even if there are no matches in the left table.
- **FULL JOIN:** returns rows when there is a match in one of the tables.
- **SELF JOIN:** is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- **CARTESIAN JOIN:** returns the Cartesian product of the sets of records from the two or more joined tables.

1. INNER JOIN:

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

The basic syntax of **INNER JOIN** is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_filed = table2.common_field;
```

2. LEFT JOIN

The SQL **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in right table, the join will still return a row in the result, but with NULL in each column from right table. This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

The basic syntax of **LEFT JOIN** is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1
LEFT JOIN table2
ON table1.common_filed = table2.common_field;
```

3. RIGHT JOIN:

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

The basic syntax of **RIGHT JOIN** is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1
RIGHT JOIN table2
ON table1.common_filed = table2.common_field;
```

4. FULL JOIN:

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

The basic syntax of **FULL JOIN** is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1
FULL JOIN table2
ON table1.common_filed = table2.common_field;
```

5. SELF JOIN:

The SQL **SELF JOIN** is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

The basic syntax of **SELF JOIN** is as follows:

```
SELECT a.column_name, b.column_name...
FROM table1 a, table1 b
WHERE a.common_filed = b.common_field;
```

6. CARTESIAN JOIN

The **CARTESIAN JOIN** or **CROSS JOIN** returns the cartesian product of the sets of records from the two or more joined tables. Thus, it equates to an inner join where the join-condition always evaluates to True or where the join-condition is absent from the statement.

The basic syntax of **INNER JOIN** is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1, table2 [, table3 ]
```

SQL SUBQUERY:

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause. A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

There are a few rules that subqueries must follow:

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

Subqueries with the SELECT Statement
Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows:

```
SELECT column_name [, column_name ]
FROM table1 [, table2 ]
WHERE column_name OPERATOR
(SELECT column_name [, column_name ]
FROM table1 [, table2 ]
[WHERE])
```

Example
Consider the CUSTOMERS table having the following records.

```
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 35 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
```

| 7 | Muffy | 24 | Indore | 10000.00 |
+----+----------+-----+-----------+----------+

Now, let us check the following subquery with a SELECT statement.

```
SQL> SELECT *
FROM CUSTOMERS
WHERE ID IN (SELECT ID
FROM CUSTOMERS
WHERE SALARY > 4500) ;
```

This would produce the following result.

```
+----+----------+-----+---------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+---------+----------+
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
+----+----------+-----+---------+----------+
```

Subqueries with the INSERT Statement

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions.

The basic syntax is as follows.

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
SELECT [ *|column1 [, column2 ]
FROM table1 [, table2 ]
[ WHERE VALUE OPERATOR ]
```

Example

Consider a table CUSTOMERS_BKP with similar structure as CUSTOMERS table. Now to copy the complete CUSTOMERS table into the CUSTOMERS_BKP table, you can use the following syntax.

```
SQL> INSERT INTO CUSTOMERS_BKP
SELECT * FROM CUSTOMERS
WHERE ID IN (SELECT ID
FROM CUSTOMERS) ;
```

Subqueries with the UPDATE Statement

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.
The basic syntax is as follows:

```
UPDATE table
SET column_name = new_value
```

[WHERE OPERATOR [VALUE]
(SELECT COLUMN_NAME
FROM TABLE_NAME)
[WHERE)]

Example

Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table. The following example updates SALARY by 0.25 times in the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

```
SQL> UPDATE CUSTOMERS
SET SALARY = SALARY * 0.25
WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
WHERE AGE >= 27 );
```

This would impact two rows and finally CUSTOMERS table would have the following records.

```
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 35 | Ahmedabad | 125.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 2125.00 |
| 6 | Komal | 22 | MP | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
+----+----------+-----+-----------+----------+
```

Subqueries with the DELETE Statement

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

The basic syntax is as follows.

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
(SELECT COLUMN_NAME
FROM TABLE_NAME)
[ WHERE) ]
```

Example

Assuming, we have a CUSTOMERS_BKP table available which is a backup of the CUSTOMERS table. The following example deletes the records from the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

```
SQL> DELETE FROM CUSTOMERS
WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
WHERE AGE >= 27 );
```

This would impact two rows and finally the CUSTOMERS table would have the following records.

```
+----+----------+-----+---------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+---------+----------+
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
+----+----------+-----+---------+----------+
```

Views in SQL:

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query. A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.
Views, which are kind of virtual tables, allow users to do the following:

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data such that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

Creating Views:

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables, or another view. To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic CREATE VIEW syntax is as follows:

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

You can include multiple tables in your SELECT statement in very similar way as you use them in normal SQL SELECT query.

Updating a View:

A view can be updated under certain conditions:

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.

- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So if a view satisfies all the above mentioned rules then you can update a view. Following is an example to update the age of Ramesh:

Inserting Rows into a View:
Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command.

Deleting Rows into a View:
Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Dropping Views:
Obviously, where you have a view, you need a way to drop the view if it is no longer needed.

The syntax is very simple as given below:

```
DROP VIEW view_name;
```

Conclusion:
We studied suitable database application using SQL DML statements: all types of Join, Sub-Query and View.

Questions

1. Explain set membership and set comparison operators?
2. Explain use of joins with example?
3. Explain set membership and set comparison operators?
4. Consider relational schema
   Employee (Empno, EName , DeptNo, Salary),
   Department(DeptNo, DName) Write SQL Queries for
   following questions (Any two)
   i) List Employee Names of 'Computer' Department.
   ii) Find average salary of each department.
   iii) Find Department name of employee name 'Amit'
5. Consider following database
   Cricket_player(p_id, Name, Address)
   Matches(Match_code, match_date, match_place)
   Score(p_id, match_code, score)
   Write following queries in SQL
   i) List player name, match_date, match_place and score of each player.
   ii) List all those players, whose maximum score is higher than 50.

Assignment No: 4

Title: Unnamed PL/SQL code block: Use of Control structure and Exception handling is mandatory. Write a PL/SQL block of code for the following requirements:-
Schema:
1. Borrower(Rollin, Name, DateofIssue, NameofBook, Status)
2. Fine(Roll_no,Date,Amt)

- Accept roll_no & name of book from user.
- Check the number of days (from date of issue), if days are between 15 to 30 then fine amount will be Rs 5per day.
- If no. of days>30, per day fine will be Rs 50 per day & for days less than 30, Rs. 5 per day.
- After submitting the book, status will change from I to R.
- If condition of fine is true, then details will be stored into fine table.

OR

Write a PL/SQL code block to calculate the area of a circle for a value of radius varying from 5 to 9. Store the radius and the corresponding values of calculated area in an empty table named areas, consisting of two columns, radius and area.

Requirements:

1. Computer System with Linux/Open Source Operating System.
2. Mysql Server

Theory:

PL/SQL is Oracle's procedural language extension to SQL. PL/SQL allows you to mix SQL statements with procedural statements like IF statement, Looping structures etc.

It is extension of SQL the following or advantages of PL/SQL.

1. We can use programming features like if statement loops etc.
2. PL/SQL helps in reducing network traffic.
3. We can have user defined error massages by using concept of exception handling.
4. We can perform related actions by using concept of Triggers.
5. We can save the source code permanently for repeated execution.

PL/SQL Block:

A PL/SQL programs called as PL/SQL block.

PL/SQL Block:

DECLARE

Declaration of variable

Declaration of cursor---------- (OPTIONAL)

Declaration of exception

BEGIN

Executable commands--------- (MANDATORY)

EXCEPTION

Exception handlers---------- (OPTIONAL)

END; / To execute the program / command

Declare: This section is used to declare local variables, cursors, Exceptions and etc. This section is optional.

Executable Section: This section contains lines of code which is used to complete table. It is mandatory.

Exception Section: This section contains lines of code which will be executed only when exception is raised. This section is optional.

Simplest PL/SQL Block:

Begin

END;

SERVEROUTPUT

This will be used to display the output of the PL/SQL programs. By default this will be off.

Syntax: Set serveroutput on | off

Ex: SQL> set serveroutput on

Conclusion :

Hence we have studied the concept of PL-SQL .

Questions:

1. What is need of PL-SQL?
2. What is the Block structure of PL-SQL block?
3. How to handle exception handling in PL-SQL block?
4. How to write a code of PL-SQL and execute it?
5. Write a PL/SQL block for following requirement and handle the exceptions.
   Roll no. of student will be entered by user. Attendance of roll no. entered by user will be checked in Student table. If attendance is less than 75% then display the message “Term not granted” and set the status in Student table as “D”. Otherwise display message “Term granted” and set the status in Student table as “ND”.

Experiment No-6

Title : Named PL/SQL Block: PL/SQL Stored Procedure and Stored Function.

Write a Stored Procedure namely proc_Grade for the categorization of student. If marks scored by students in examination is <=1500 and marks>=990 then student will be placed in distinction category if marks scored are between 989 and900 category is first class, if marks 899 and 825 category is Higher Second Class

Write a PL/SQL block for using procedure created with above requirement. Stud_Marks(name, total_marks) Result(Roll,Name, Class)

Theory :

- **Stored Procedures**

A **stored procedure** or in simple a **proc** is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages.

A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block.

A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.

Procedures: Passing Parameters

We can pass parameters to procedures in three ways.

1) IN-parameters

2) OUT-parameters

3) IN OUT-parameters

A procedure may or may not return any value.

General Syntax to create a procedure is:

CREATE [OR REPLACE] PROCEDURE proc_name [list of parameters]

IS

Declaration section

BEGIN

Execution section

EXCEPTION

Exception section

END;

IS - marks the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

The syntax within the brackets [ ] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

Procedures: Example

The below example creates a procedure 'employer_details' which gives the details of the employee.

```
CREATE OR REPLACE PROCEDURE employer_details
IS
CURSOREmp_cur IS
SELECTfirst_name, last_name, salary FROM emp_tbl;
emp_recemp_cur%rowtype;
BEGIN
FOREmp_rec in sales_cur
LOOP
dbms_output.put_line(emp_cur.first_name || ' ' ||emp_cur.last_name || ' ' ||emp_cur.salary);
END LOOP;
END;
/
```

How to execute a Stored Procedure?

There are two ways to execute a procedure.

1) From the SQL prompt.

```
EXECUTE [or EXEC] procedure_name;
```

2) Within another procedure – simply use the procedure name.

```
procedure_name;
```

NOTE: In the examples given above, we are using backward slash '/' at the end of the program. This indicates the oracle engine that the PL/SQL program has ended and it can begin processing the statements.

- PL/SQL Functions

A function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value.

General Syntax to create a function is

CREATE [OR REPLACE] FUNCTION function_name [parameters]

RETURN return_datatype;

IS

Declaration_section

BEGIN

Execution_section

Return return_variable;

EXCEPTION

exception section

Return return_variable;

END;

1) **Return Type:** The header section defines the return type of the function. The return datatype can be any of the oracle datatype like varchar, number etc.

2) The execution and exception section both should return a value which is of the datatype defined in the header section.

For example, let's create a frunction called "employer_details_func' similar to the one created in stored proc

```
CREATE OR REPLACE FUNCTION employer_details_func
 RETURN VARCHAR(20);
IS
emp_nameVARCHAR(20);
BEGIN
SELECT first_name INTO emp_name
FROM emp_tbl WHERE empID = '100';
RETURN emp_name;
 END;
```

/

In the example we are retrieving the ‘first_name’ of employee with empID 100 to variable ‘emp_name’.

The return type of the function is VARCHAR which is declared in line no 2. The function returns the 'emp_name' which is of type VARCHAR as the return value in line no 9.

How to execute a PL/SQL Function?

A function can be executed in the following ways.

1) Since a function returns a value we can assign it to a variable.

employee_name := employer_details_func;

If ‘employee_name’ is of datatype varchar we can store the name of the employee by assigning the return type of the function to it.

2) As a part of a SELECT statement

SELECT employer_details_func FROM dual;

3) In a PL/SQL Statements like,

dbms_output.put_line(employer_details_func);

This line displays the value returned by the function.

Conclusion :

Hence we have studied aPL/SQL Stored Procedure and Stored Function.

Questions:

1. What are the differences between procedure and function?
2. Explain syntax of stored procedure and how to execute it?
3. Explain syntax of function and how to execute it?
4. Explain in and out parameters?
5. What is a need of procedure and function?

Experiment No: 6

Title: Cursors: (All types: Implicit, Explicit, Cursor FOR Loop, Parameterized Cursor)

Write a PL/SQL block of code using parameterized Cursor, that will merge the data available in the newly created table N_RollCall with the data available in the table O_RollCall. If the data in the first table already exist in the second table then that data should be skipped.

Theory :

A cursor is a temporary work area created in the system memory when a SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it.

This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the active set.

There are two types of cursors in PL/SQL:

- **Implicit cursors**

These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed. They are also created when a SELECT statement that returns just one row is executed.

- **Explicit cursors**

They must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When you fetch a row the current row position moves to next row.

Both implicit and explicit cursors have the same functionality, but they differ in the way they are accessed.

- **Implicit Cursors: Application**

When you execute DML statements like DELETE, INSERT, UPDATE and SELECT statements, implicit statements are created to process these statements.

Oracle provides few attributes called as implicit cursor attributes to check the status of DML operations. The cursor attributes available are %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN.

For example, When you execute INSERT, UPDATE, or DELETE statements the cursor attributes tell us whether any rows are affected and how many have been affected.
When a SELECT... INTO statement is executed in a PL/SQL Block, implicit cursor attributes can be used to find out whether any row has been returned by the SELECT statement. PL/SQL returns an error when no data is selected.
The status of the cursor for each of these attributes are defined in the below table.

Attributes	**Return Value**	**Example**
%FOUND	The return value is TRUE, if the DML statements like INSERT, DELETE and UPDATE affect at least one row and if SELECT ….INTO statement return at least one row.	SQL%FOUND
	The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE do not affect row and if SELECT….INTO statement do not return a row.	
%NOTFOUND	The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE at least one row and if SELECT ….INTO statement return at least one row.	SQL%NOTFOUND
	The return value is TRUE, if a DML statement like INSERT, DELETE and UPDATE do not affect even one row and if SELECT ….INTO statement does not return a row.	
%ROWCOUNT	Return the number of rows affected by the DML operations INSERT, DELETE, UPDATE, SELECT	SQL%ROWCOUNT

For Example: Consider the PL/SQL Block that uses implicit cursor attributes as shown below:

```
DECLARE  var_rows number(5);
BEGIN
  UPDATE employee
  SET salary = salary + 1000;
  IF SQL%NOTFOUND THEN
dbms_output.put_line('None of the salaries where updated');
  ELSIF SQL%FOUND THEN
var_rows := SQL%ROWCOUNT;
dbms_output.put_line('Salaries for ' || var_rows || 'employees are updated');
  END IF;
END;
```

In the above PL/SQL Block, the salaries of all the employees in the ‘employee’ table are updated. If none of the employee’s salary are updated we get a message 'None of the salaries where updated'. Else we get a message like for example, 'Salaries for 1000 employees are updated' if there are 1000 rows in ‘employee’ table.

Conclusion:

Thus, we have studied Implicit, Explicit, Cursor FOR Loop, and Parameterized Cursor.

Questions:

1. What is a need of cursor?
2. What are different types of cursor?
3. What is Implicit and Explicit cursor?
4. How to write code of cursor and how to execute?
5. Explain Parameterized cursor?

Experiment No.: 7

Title: Database Trigger (All Types: Row level and Statement level triggers, Before and After Triggers). Write a database trigger on Library table. The System should keep track of the records that are being updated or deleted. The old value of updated or deleted records should be added in Library_Audit table.

Requirements:

1. Computer System with Linux/Open Source Operating System.
2. Sql Server

Theory:

A trigger is a pl/sql block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed.

Syntax for Creating a Trigger

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
BEGIN
 --- sql statements
END;
```

- CREATE [OR REPLACE] TRIGGER trigger_name - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- {BEFORE | AFTER | INSTEAD OF } - This clause indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. before and after cannot be used to create a trigger on a view.

- {INSERT [OR] | UPDATE [OR] | DELETE} - This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.
- [OF col_name] - This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.
- CREATE [OR REPLACE] TRIGGER trigger_name - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- [ON table_name] - This clause identifies the name of the table or view to which the trigger is associated.
- [REFERENCING OLD AS o NEW AS n] - This clause is used to reference the old and new values of the data being changed. By default, you reference the values as :old.column_name or :new.column_name. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.
- [FOR EACH ROW] - This clause is used to determine whether a trigger must fire when each row gets affected (i.e. a Row Level Trigger) or just once when the entire sql statement is executed(i.e.statement level Trigger).
- WHEN (condition) - This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

For Example: The price of a product changes constantly. It is important to maintain the history of the prices of the products.

We can create a trigger to update the 'product_price_history' table when the price of the product is updated in the 'product' table.

1) Create the 'product' table and 'product_price_history' table

```
CREATE TABLE product_price_history
(product_id number(5),
product_namevarchar2(32),
supplier_namevarchar2(32),
unit_price number(7,2) );
```

```
CREATE TABLE product
(product_id number(5),
product_namevarchar2(32),
supplier_namevarchar2(32),
unit_price number(7,2) );
```

2) Create the price_history_trigger and execute it.

```
CREATE or REPLACE TRIGGER price_history_trigger
BEFORE UPDATE OF unit_price
ON product
FOR EACH ROW
BEGIN
INSERT INTO product_price_history
VALUES
(:old.product_id,
:old.product_name,
:old.supplier_name,
:old.unit_price);
END;
/
```

3) Lets update the price of a product.

```
UPDATE PRODUCT SET unit_price = 800 WHERE product_id = 100
```

Once the above update query is executed, the trigger fires and updates the 'product_price_history' table.

4)If you ROLLBACK the transaction before committing to the database, the data inserted to the table is also rolled back.

- **Types of PL/SQL Triggers**

There are two types of triggers based on the which level it is triggered.

1) Row level trigger - An event is triggered for each row upated, inserted or deleted.

2) Statement level trigger - An event is triggered for each sql statement executed.

PL/SQL Trigger Execution Hierarchy

The following hierarchy is followed when a trigger is fired.

1) **BEFORE** statement trigger fires first.

2) **Next BEFORE** row level trigger fires, once for each row affected.

3) **Then AFTER** row level trigger fires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.

4) Finally the **AFTER** statement level trigger fires.

For Example: Let's create a table 'product_check' which we can use to store messages when triggers are fired.

```
CREATE TABLE product
(Message varchar2(50),
Current_Datenumber(32)
);
```

Let's create a BEFORE and AFTER statement and row level triggers for the product table.

1) BEFORE UPDATE, Statement Level: This trigger will insert a record into the table 'product_check' before a sql update statement is executed, at the statement level.

```
CREATE or REPLACE TRIGGER Before_Update_Stat_product
BEFORE
UPDATE ON product
Begin
INSERT INTO product_check
Values('Before update, statement level',sysdate);
END;
/
```

2) BEFORE UPDATE, Row Level: This trigger will insert a record into the table 'product_check' before each row is updated.

```
 CREATE or REPLACE TRIGGER Before_Upddate_Row_product
 BEFORE
 UPDATE ON product
 FOR EACH ROW
```

```
BEGIN
INSERT INTO product_check
Values('Before update row level',sysdate);
END;
/
```

3) AFTER UPDATE, Statement Level: This trigger will insert a record into the table 'product_check' after a sql update statement is executed, at the statement level.

```
CREATE or REPLACE TRIGGER After_Update_Stat_product
AFTER
UPDATE ON product
BEGIN
INSERT INTO product_check
Values('After update, statement level', sysdate);
End;
/
```

4) AFTER UPDATE, Row Level: This trigger will insert a record into the table 'product_check' after each row is updated.

```
CREATE or REPLACE TRIGGER After_Update_Row_product
AFTER
insert On product
FOR EACH ROW
BEGIN
INSERT INTO product_check
Values('After update, Row level',sysdate);
END;
/
```

Now lets execute a update statement on table product.

```
UPDATE PRODUCT SET unit_price = 800
WHERE product_id in (100,101);
```

Lets check the data in 'product_check' table to see the order in which the trigger is fired.

SELECT * FROM product_check;

Output:

Mesage	Current_Date
Before update, statement level	26-Nov-2008
Before update, row level	26-Nov-2008
After update, Row level	26-Nov-2008
Before update, row level	26-Nov-2008
After update, Row level	26-Nov-2008
After update, statement level	26-Nov-2008

The above result shows 'before update' and 'after update' row level events have occured twice, since two records were updated. But 'before update' and 'after update' statement level events are fired only once per sql statement.

The above rules apply similarly for INSERT and DELETE statements.

Conclusion :

Hence we have studied Row level and Statement level triggers, Before and After Triggers

Questions:

1. Explain the concept of triggers?
2. What are the types of triggers?
3. What is a Row level and statement level trigger?
4. What is the difference between before and after trigger with example?

Experiments No.8

Title: Implement MYSQL/Oracle database connectivity with PHP/ python/Java Implement Database navigation operations (add, delete, edit,) using ODBC/JDBC.

Requirements:

1. Computer System with Linux/Open Source Operating System.
2. Mysql Server
3. JDK

Theory:

The JDBC (Java Database Connectivity) API defines interfaces and classes for writing database applications in Java by making database connections. Using JDBC you can send SQL, PL/SQL statements to almost any relational database. JDBC is a Java API for executing SQL statements and supports basic SQL functionality. It provides RDBMS access by allowing you to embed SQL inside Java code. Because Java can run on a thin client, applets embedded in Web pages can contain downloadable JDBC code to enable remote database access. You will learn how to create a table, insert values into it, query the table, retrieve results, and update the table with the help of a JDBC Program example.
Although JDBC was designed specifically to provide a Java interface to relational databases, you may find that you need to write Java code to access non-relational databases as well.
JDBC Architecture
Java application calls the JDBC library. JDBC loads a driver which talks to the database
In general, to process any SQL statement with JDBC, you follow these steps:

1) **Establishing a connection.**
2) **Create a statement.**
3) **Execute the query.**
4) **Process the *ResultSet* object.**
5) **Close the connection.**

1. Configuring a JDBC development environment

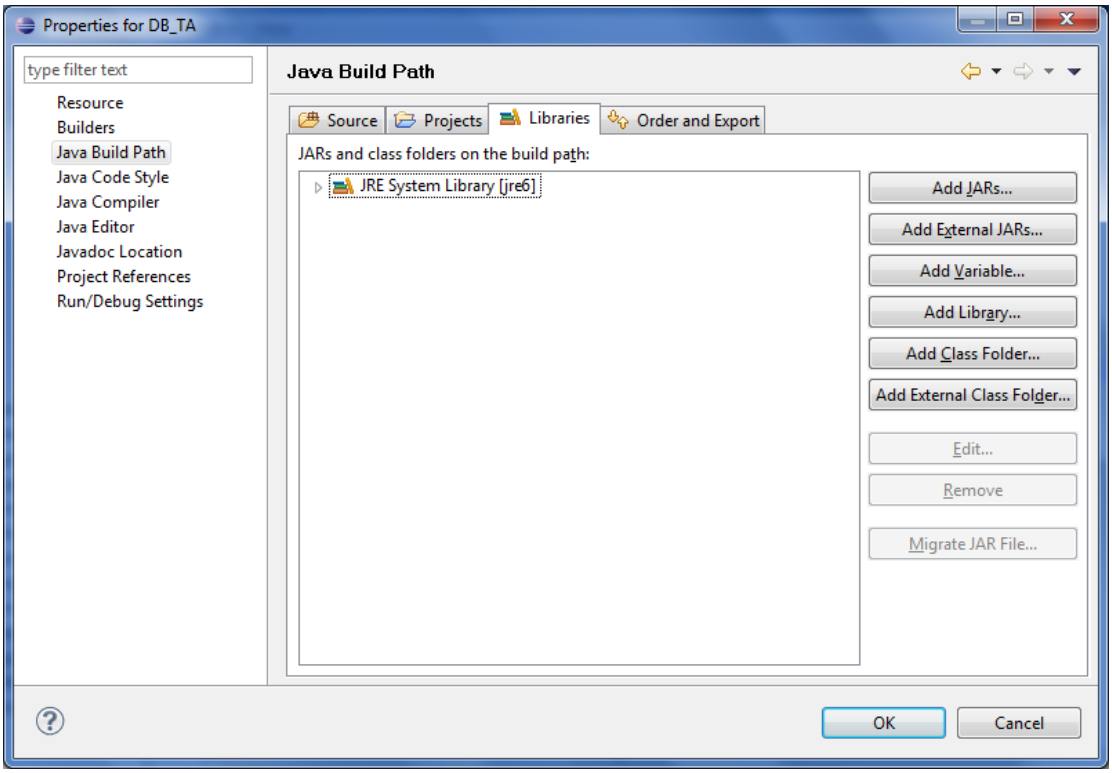
First of all, you should download the JDBC Driver for your DBMS. For this class, you can use "ojdbc6.jar" provided on class web-site. The ojdbc6.jar is a JDBC driver for Oracle Database 11g.
The below web sites are official pages for downloading official version of JDBC drivers.
Oracle : http://www.oracle.com/technetwork/database/features/jdbc/index-091264.html
Mysql : http://www.mysql.com/downloads/connector/j/

A. Install the latest version of the Java SE SDK on your computer.

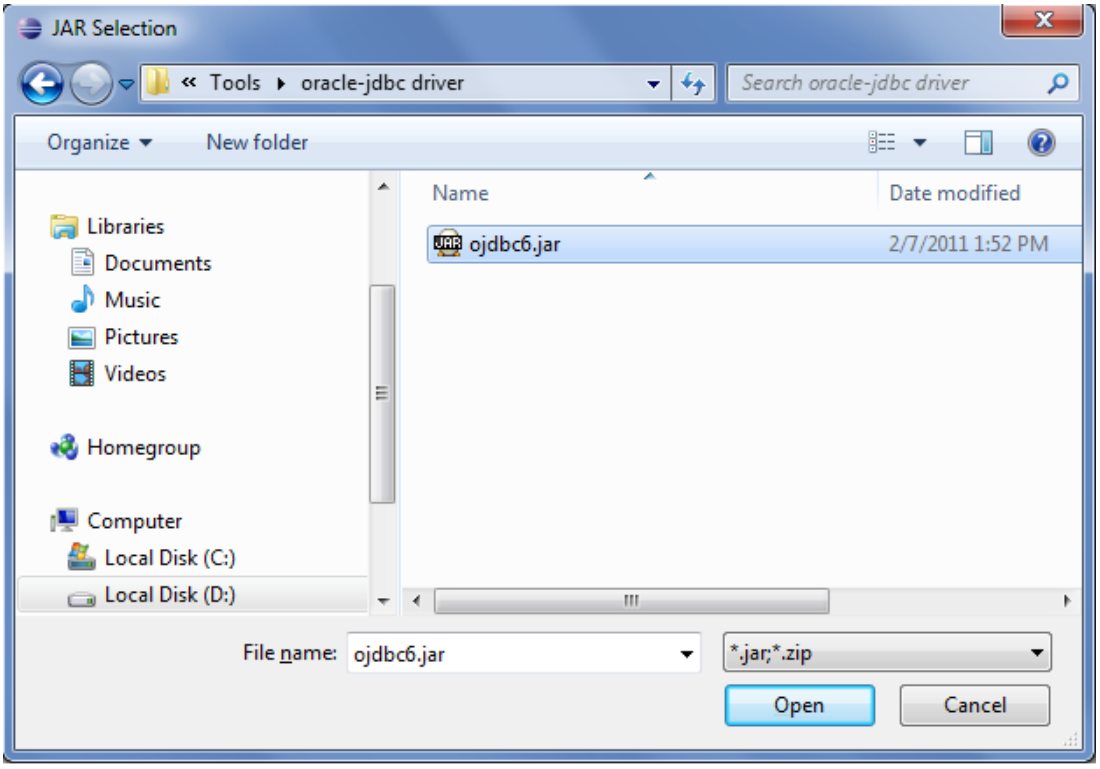
B. Set up the classpath for a JDBC driver.

The JDBC driver is not needed for compiling the java source but it is needed to execute the class. There are so many ways to set up this classpath according to the OS, programming tools and your preferences. This tutorial provides the case to use Eclipse.
If you use the eclipse, you can use following description.

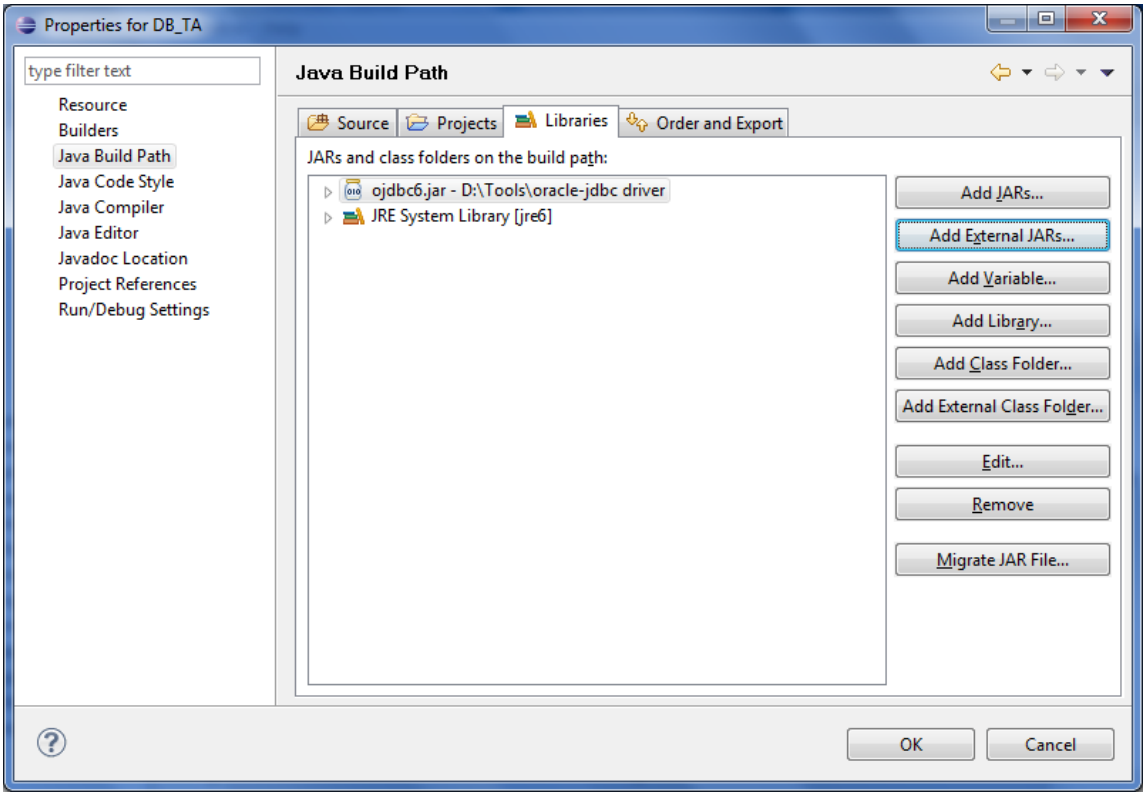
- Eclipse main menu-> Project -> Properties -> Java Build Path -> Librarie



- Click Add External JARs -> Select the JDBC driver. -> Open



- Click Add External JARs -> Select the JDBC driver. -> Open



- You can confirm the jar file is added then click OK.

2. Establishing a Connection

In this step of the jdbc connection process, we load the driver class by calling Class.forName() with the Driver class name as an argument. Once loaded, the Driver class creates an instance of itself.

Example for loading JDBC driver

```
String driverName = "oracle.jdbc.driver.OracleDriver"; // for Oracle
// String driverName = “com.mysql.jdbc.Driver”; //for MySql
try {
// Load the JDBC driver
Class.forName(driverName);
} catch (ClassNotFoundException e) {
// Could not find the database driver
System.out.println("ClassNotFoundException : "+e.getMessage());
}
```

The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager is considered the backbone of JDBC architecture. DriverManager class manages the JDBC drivers that are installed on the system. Its getConnection() method is used to establish a connection to a database. It uses a username, password, and a jdbc url to establish a connection to the database and returns a connection object. A jdbc Connection represents a session/connection with a specific database. Within the context of a Connection, SQL, PL/SQL statements are executed and results are returned. An application can have one or more connections with a single database, or it can have many connections with different databases.

Example for JDBC connection

```
String serverName = "linux.grace.umd.edu";
String portNumber = "1521";
String sid = "dbclass1";
String url = "jdbc:oracle:thin:@" + serverName + ":" +
portNumber + ":" + sid; // for Oracle
//uri =”jdbc:mysql://server ip or address:port/database name”; //for Mysql
try {
// Create a connection to the database
connection = DriverManager.getConnection(url, username, password);
catch (SQLException e) {
// Could not connect to the database
System.out.println(e.getMessage());
}
```

➢ Example for loading and connection

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class TestCon{
Connection connection = null;
String driverName ="oracle.jdbc.driver.OracleDriver"; // for Oracle
// String driverName = “com.mysql.jdbc.Driver”; //for MySql
String serverName = "ginger.umd.edu"; // Use this server.
String portNumber = "1521";
String sid = "dbclass1";
String url="jdbc:oracle:thin:@"+serverName+":"+ portNumber+":"+sid; // for Oracle
//uri =”jdbc:mysql://server ip or address:port/database name”; //for Mysql
String username = "your user name"; // You should modify this.
String password = "your password"; // You should modify this.
public TestCon() {}
public boolean doConnection(){
try {
// Load the JDBC driver
Class.forName(driverName);
// Create a connection to the database
connection = DriverManager.getConnection(url, username, password);
} catch (ClassNotFoundException e) {
// Could not find the database driver
System.out.println("ClassNotFoundException : "+e.getMessage());
return false;
} catch (SQLException e) {
```

```
// Could not connect to the database
System.out.println(e.getMessage());
return false;
}
return true;
}
public static void main(String arg[]){
TestCon con =new TestCon();
System.out.println("Connection : " +con.doConnection());
}
}
```

If the output of the previous example is “Connection : true”, your environment setting and connection is correct. If not, try to check the jdbc driver classpath, your username and your password

3. Creating statements & executing queries

A Statement is an interface that represents a SQL statement. You execute Statement objects, and they generate ResultSet objects, which is a table of data representing a database result set. You need a Connection object to create a Statement object.

There are three different kinds of statements.

- **Statement**: Used to implement simple SQL statements with no parameters.
- **PreparedStatement**: (Extends Statement.) Used for precompiling SQL statements that might contain input parameters. See Using Prepared Statements for more information.
- **CallableStatement**: (Extends PreparedStatement.) Used to execute stored procedures that may contain both input and output parameters. See Stored Procedures for more information.

➢ Example for Statement

```
Statement stmt = null;
try {
stmt = connection.createStatement();
} catch (SQLException e ) {
System.out.println(e.getMessage());
}
```

To execute a query, call an execute method from Statement such as the following:

- **execute**: Returns true if the first object that the query returns is a ResultSet object. Use this method if the query could return one or moreResultSet objects. Retrieve the ResultSet objects returned from the query by repeatedly calling Statement.getResutSet.
- **executeQuery**: Returns one ResultSet object.
- **executeUpdate**: Returns an integer representing the number of rows affected by the SQL statement. Use this method if you are using INSERT,DELETE, or UPDATE SQL statements.

➢ Example for executeQuery

```
String country=”D”;
Statement stmt = null;
String query = " SELECT * FROM CITY WHERE country=”’+country+”’”;
try {
stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
String name = rs.getString(1); // or rs.getString("NAME");
```

```
String coun= rs.getString(2);
String province = rs.getString(3);
int population = rs.getInt(4);
}
stmt.close();
} catch (SQLException e ) {
System.out.println(e.getMessage());
}
```
The re.next() return „true“ until there is no more result.

- **Example for executeQuery**

```
String population=”1705000”;
String cityName=”Hamburg”;
String province=”Hamburg”;
Statement stmt = null;
try {
stmt = connection.createStatement();
String sql = "UPDATE CITY SET population=”+ population +”" WHERE NAME=”+ cityName +”"
AND PROVINCE=“”+ province +””“"; stmt.executeUpdate(sql);s
stmt.close();
} catch (SQLException e ) {
System.out.println(e.getMessage());
}
```

Example of simple JDBC program

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class TestCon{
Connection connection = null;
String driverName ="oracle.jdbc.driver.OracleDriver"; // for Oracle
// String driverName = “com.mysql.jdbc.Driver”; //for MySql
String serverName = "ginger.umd.edu";
String portNumber = "1521";
String sid = "dbclass1";
String url = "jdbc:oracle:thin:@" + serverName + ":" + portNumber + ":" + sid; // for Oracle
//uri =”jdbc:mysql://server ip or address:port/database name”; //for Mysql
String username = "XXXXXXXX"; //your username
String password = "XXXXXXXX"; //your password
public TestCon() {
}
public boolean doConnection(){
try {
// Load the JDBC driver
Class.forName(driverName);
// Create a connection to the database
connection = DriverManager.getConnection(url, username, password);
} catch (ClassNotFoundException e) {
```

```
// Could not find the database driver
System.out.println("ClassNotFoundException : "+e.getMessage());
return false;
} catch (SQLException e) {
// Could not connect to the database
System.out.println(e.getMessage());
return false;
}
return true;
}
public void printCounryByCapital(String capital) throws SQLException{
Statement stmt = null;
String query = "SELECT * FROM COUNTRY WHERE CAPITAL='"+capital+"'";
stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
String name = rs.getString(1); // or rs.getString("NAME");
String code= rs.getString(2);
String cap = rs.getString(3);
String province = rs.getString(4);
int area = rs.getInt(5);
int population = rs.getInt(6);
System.out.println(" Name : "+name);
System.out.println(" Code : "+code);
System.out.println(" Capital : "+cap);
System.out.println(" Province : "+province);
System.out.println(" Area : "+area);
System.out.println(" Population : "+population);
}
}
public void printCityByCountry(String country) throws SQLException{
Statement stmt = null;
String query = " SELECT * FROM CITY WHERE country='"+country+"'";
stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
String name = rs.getString(1); // or rs.getString("NAME");
String coun= rs.getString(2);
String province = rs.getString(3);
int population = rs.getInt(4);
System.out.println(" Name : "+name);
System.out.println(" Country : "+coun);
System.out.println(" Province : "+province);
System.out.println(" Population : "+population);
}
stmt.close();
}
public void updateCityPopulation(String cityName,String province,
String population)throws SQLException
{
Statement stmt = null;
```

```
stmt = connection.createStatement();
String sql = "UPDATE CITY SET population='"+ population
+"' WHERE NAME='"+cityName +"' AND PROVINCE='"+ province +"'";
stmt.executeUpdate(sql);
stmt.close();
}
public static void main(String arg[]){
TestCon con =new TestCon();
System.out.println("Connection : " +con.doConnection());
try{
con.printCounryByCapital("Paris");
con.printCityByCountry("D");
con.updateCityPopulation("Munich","Bayern","3000");
}catch(SQLException ex){System.out.println(ex.getMessage());}
}
}
```

Conclusion:

We studied database MySQL-JAVA Connectivity.

Questions:

1. What are functions of jdbc driver?
2. What are the steps to connect to the database in java?
3. What are the JDBC statements?
4. What the differences are between execute, executeQuery, and executeUpdate?
5. What does the JDBC ResultSet interface?

Experiment No.9

Aim: Design and Develop MongoDB Queries using CRUD operations. (Use CRUD operations, SAVE method, logical operators)

Requirements:

1. Computer System with Linux/Open Source Operating System.
2. Mongodb Server

Theory:

MongoDB use DATABASE_NAME is used to create database. The command will create a new database if it doesn't exist, otherwise it will return the existing database

Syntax

Basic syntax of use DATABASE statement is as follows –

use DATABASE_NAME

Example

If you want to create a database with name <mydb>, then use DATABASE statement would be as follows –

>use mydb

switched to dbmydb

To check your currently selected database, use the command db

>dbmydb

If you want to check your databases list, use the command show dbs.

>show dbs

local 0.78125GB

test 0.23012GB

Your created database (mydb) is not present in list. To display database, you need to insert at least one document into it

>db.movie.insert({"name":"tutorials point"})

>show dbs

local 0.78125GB

mydb 0.23012GB

test 0.23012GB

In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.

The dropDatabase() Method

MongoDB db.dropDatabase() command is used to drop a existing database.

Syntax

Basic syntax of dropDatabase() command is as follows –

db.dropDatabase()

This will delete the selected database. If you have not selected any database, then it will delete default 'test' database.

Example

First, check the list of available databases by using the command, show dbs.

```
>show dbs
local      0.78125GB
mydb       0.23012GB
test       0.23012GB
```

If you want to delete new database <mydb>, then dropDatabase() command would be as follows –

```
>use mydb
switched to dbmydb
>db.dropDatabase()
>{ "dropped" : "mydb", "ok" : 1 }
>
```

Now check list of databases.

```
>show dbs
local      0.78125GB
test       0.23012GB
>
```

The createCollection() Method

MongoDB db.createCollection(name, options) is used to create collection.

Syntax

Basic syntax of createCollection() command is as follows –

db.createCollection(name, options)

In the command, name is name of collection to be created. Options is a document and is used to specify configuration of collection.

Examples

Basic syntax of createCollection() method without options is as follows –

```
>use test
switched to db test
>db.createCollection("mycollection")
{ "ok" : 1 }
>
```

You can check the created collection by using the command show collections.

```
>show collections
mycollection
system.indexes
```

The following example shows the syntax of createCollection() method with few important options −

```
>db.createCollection("mycol", { capped : true, autoIndexId : true, size :
   6142800, max : 10000 } )
{ "ok" : 1 }
>
```

The drop() Method

MongoDB's db.collection.drop() is used to drop a collection from the database.

Syntax

Basic syntax of drop() command is as follows −

```
db.COLLECTION_NAME.drop()
```

The insert() Method

To insert data into MongoDB collection, you need to use MongoDB's insert() or save() method.

Syntax

The basic syntax of insert() command is as follows −

```
>db.COLLECTION_NAME.insert(document)
```

Example

```
>db.mycol.insert({
   _id: ObjectId(7df78ad8902c),
title: 'MongoDB Overview',
description: 'MongoDB is no sql database',
by: 'tutorials point',
url: 'http://www.tutorialspoint.com',
tags: ['mongodb', 'database', 'NoSQL'],
likes: 100
})
```

MongoDB Update() Method

The update() method updates the values in the existing document.

Syntax

The basic syntax of update() method is as follows −

```
>db.COLLECTION_NAME.update(SELECTION_CRITERIA, UPDATED_DATA)
```

Example

Consider the mycol collection has the following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Following example will set the new title 'New MongoDB Tutorial' of the documents whose title is 'MongoDB Overview'.

```
>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}})
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"New MongoDB Tutorial"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
>
```

By default, MongoDB will update only a single document. To update multiple documents, you need to set a parameter 'multi' to true.

```
>db.mycol.update({'title':'MongoDB Overview'},
  {$set:{'title':'New MongoDB Tutorial'}},{multi:true})
```

MongoDB Save() Method

The save() method replaces the existing document with the new document passed in the save() method.

Syntax

The basic syntax of MongoDB save() method is shown below −

```
>db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})
```

Example

Following example will replace the document with the _id '5983548781331adf45ec7'.

```
>db.mycol.save(
  {
    "_id" :ObjectId(5983548781331adf45ec7), "title":"Tutorials Point New Topic",
      "by":"Tutorials Point"
  }
)
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"Tutorials Point New Topic",
  "by":"Tutorials Point"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
>
```

By default, MongoDB will update only a single document. To update multiple documents, you need to set a parameter 'multi' to true.

```
>db.mycol.update({'title':'MongoDB Overview'},
  {$set:{'title':'New MongoDB Tutorial'}},{multi:true})
```

- **Use of Operators**

1. gt – Greater Than

   ```
   db.cdatail1.find({pop:{$gt:10000000}})
   ```

2. gte – Greater Than equal to

   db.cdatail1.find({pop:{$gte:10000000}})
3. lt – less Than

   db.cdatail1.find({pop:{$lt:10000000}})
4. lte – less Than equal to

   db.cdatail1.find({pop:{$lte:10000000}})
5. ne – not equal to

   db.cdatail1.find({state:{$ne:"AP"}})
6. in – present in set

   db.cdatail1.find({state:{$in:["AP","MH"]}})
7. nin – not present in set

   db.cdatail1.find({state:{$nin:["AP","MH"]}})
8. or- logical operator

   db.cdatail1.find({$or:[{state:{$ne:"MH"}},{city:{$ne:"Pune"}}]})
9. and- logical operator

   db.cdatail1.find({$or:[{state:{$ne:"MH"}},{pop:{$gt:100000000}}]})

Conclusion: Hence we study CRUD operations, SAVE method, logical operators.

Questions:

1. Explain how to use save method with example?
2. Explain Logical operators?
3. Consider schema User (user_id, age, status). Write MongoDB Schema statements for following queries
   i) Create Collection and Document
   ii) Insert Data and Update Document
   iii) Find all the users whose age is equal to 50 or status is “A”
   iv) Update the user’s age increment by 3 whose status is “A”
   v) Delete the users whose status is “A”
4. Consider following structure for MongoDB collection and write a query for following requirements in MongoDB
   Teachers(Tname, dno, experience, salary, date_of_joining)
   Department(Dno, Dname)

Students(Sname, roll_no, class)

i) Write a query to create above collection & for insertion of some sample documents.

ii) Find the information about all teachers of dno = 2 and having salary greater than or equal to 10,000/-

iii) Find the student information having roll_no=2 or Sname = xyz .

Experiment No.10

Title: Implement aggregation and indexing with suitable example using MongoDB

Requirements:

1. Computer System with Linux/Open Source Operating System.
2. Mongodb Server

Theory :

MongoDB Aggregation:

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. In sql count(*) and with group by is an equivalent of mongodb aggregation.

The aggregate() Method:

For the aggregation in mongodb you should use **aggregate()** method.

Syntax: >db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)

Example

In the collection you have the following data −

```
{
   _id: ObjectId(7df78ad8902c)
title: 'MongoDB Overview',
description: 'MongoDB is no sql database',
by_user: 'tutorials point',
url: 'http://www.tutorialspoint.com',
tags: ['mongodb', 'database', 'NoSQL'],
likes: 100
},
{
   _id: ObjectId(7df78ad8902d)
title: 'NoSQL Overview',
description: 'No sql database is very fast',
by_user: 'tutorials point',
```

url: 'http://www.tutorialspoint.com',
tags: ['mongodb', 'database', 'NoSQL'],
likes: 10
},
{
_id: ObjectId(7df78ad8902e)
title: 'Neo4j Overview',
description: 'Neo4j is no sql database',
by_user: 'Neo4j',
url: 'http://www.neo4j.com',
tags: ['neo4j', 'database', 'NoSQL'],
likes: 750
},

Now from the above collection, if you want to display a list stating how many tutorials are written by each user, then you will use the following **aggregate()** method

```
>db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : 1}}}])
{
  "result" : [
    {
      "_id" : "tutorials point",
      "num_tutorial" : 2
    },
    {
      "_id" : "Neo4j",
      "num_tutorial" : 1
    }
  ],
  "ok" : 1
}
```

Sql equivalent query for the above use case will be **select by_user, count(*) from mycol group by by_user**.In the above example, we have grouped documents by field **by_user** and on each

occurrence of by_user previous value of sum is incremented. Following is a list of available aggregation expressions.

Expression	Description	Example
$sum	Sums up the defined value from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : "$likes"}}}])
$avg	Calculates the average of all given values from all documents in the collection	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$avg : "$likes"}}}])
$min	Gets the minimum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$min : "$likes"}}}])
$max	Gets the maximum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$max : "$likes"}}}])
$push	Gets the maximum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$max : "$likes"}}}])
$addToSet	Inserts the value to an array in the resulting document but does not create duplicates.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$addToSet : "$url"}}}])
$first	Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage.	db.mycol.aggregate([{$group : {_id : "$by_user", first_url : {$first : "$url"}}}])
$last	Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage.	db.mycol.aggregate([{$group : {_id : "$by_user", last_url : {$last : "$url"}}}])

- **Indexing**

Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. This scan is highly inefficient and require MongoDB to process a large volume of data.

Indexes are special data structures, that store a small portion of the data set in an easy-to-traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index.

The ensureIndex() Method

To create an index you need to use ensureIndex() method of MongoDB.

Syntax

The basic syntax of ensureIndex() method is as follows().

>db.COLLECTION_NAME.ensureIndex({KEY:1})

Here key is the name of the field on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

Example

>db.mycol.ensureIndex({"title":1})

In ensureIndex() method you can pass multiple fields, to create index on multiple fields.

>db.mycol.ensureIndex({"title":1,"description":-1})

Conclusion : Hence we have studied aggregation and indexing with suitable example using MongoDB

Questions:

1. Explain different types of Aggregation method?
2. What is Ensure Indexing?
3. Explain following function

i) addToSet ii) push iii) First iv) Last

4. What is an advantage of indexing?

Experiments No.11

Aim: Implement Map reduces operation with suitable example using MongoDB

Requirements:

1. Computer System with Linux/Open Source Operating System.
2. Mongodb Server

Theory:

MongoDB Map Reduce:

As per the MongoDB documentation, Map-reduce is a data processing paradigm for condensing large volumes of data into useful aggregated results. MongoDB uses mapReduce command for map-reduce operations. MapReduce is generally used for processing large data sets.

The map-reduce function first queries the collection, then maps the result documents to emit key-value pairs which is then reduced based on the keys that have multiple values.

- **map** is a javascript function that maps a value with a key and emits a key-valur pair
- **reduce** is a javscript function that reduces or groups all the documents having the same key
- **out** specifies the location of the map-reduce query result
- **query** specifies the optional selection criteria for selecting documents
- **sort** specifies the optional sort criteria
- **limit** specifies the optional maximum number of documents to be returned

MapReduce Command:

Following is the syntax of the basic mapReduce command:

```
>db.collection.mapReduce(
function() {emit(key,value);}, //map function
function(key,values) {return reduceFunction}, //reduce function
{
out: collection,
query: document,
sort: document,
limit: number
}
```

)

Example:

Consider the following document structure storing user posts. The document stores user_name of the user and the status of post.

```
{
"post_text": "NileshKorade is an awesome website for tutorials",
"user_name": "mark",
"status":"active"
}
```

Now, we will use a mapReduce function on our **posts** collection to select all the active posts, group them on the basis of user_name and then count the number of posts by each user using the following code:

```
>db.posts.mapReduce(
function() { emit(this.user_id,1); },
function(key, values) {return Array.sum(values)},
{
query:{status:"active"},
out:"post_total"
}
)
```

Result

```
{
"result" : "post_total",
"timeMillis" : 9,
"counts" : {
"input" : 4,
"emit" : 4,
"reduce" : 2,
"output" : 2
},
"ok" : 1,
```

}

The result shows that a total of 4 documents matched the query (status:"active"), the map function emitted 4 documents with key-value pairs and finally the reduce function grouped mapped documents having the same keys into 2.

To see the result of this mapReduce query use the find operator:

```
>db.posts.mapReduce(
function() { emit(this.user_id,1); },
function(key, values) {return Array.sum(values)},
{
query:{status:"active"},
out:"post_total"
}
).find()
```

Result: The above query gives the following result which indicates that both users tom and mark have two posts in active states:

```
{"_id" : "tom", "value" : 2 }
{ "_id" : "mark", "value" : 2 }
```

Conclusion: Here we performed Map reduce operation with suitable example using MongoDB.

Questions:

1. Explain the concept of Big Data?
2. What is the concept of Map-Reduce?
3. What is a use of Map reduce()?
4. What will be the output of map()?

Experiment No:12

Title: Write a program to implement Mongo DB database connectivity with any front end language to implement Database navigation operations(add, delete, edit etc.)

Requirements:

1. Computer System with Linux/Open Source Operating System.
2. Mongodb Server

Theory :

- **Installation**

Before we start using MongoDB in our Java programs, we need to make sure that we have MongoDB JDBC Driver and Java set up on the machine. You need to download the jar from the path

"http://central.maven.org/maven2/org/mongodb/mongo-java-driver/2.11.3

You need to include the mongo.jar into your classpath.

Connect to database:

To connect database, you need to specify database name, if database doesn't exist then mongodb creates it automatically.

Classes in mongo-java-driver.jar:

1.DB:(public abstract class DB extends Object)

Package: com.mongodb.DB

An abstract class that represents a logical database on a server

Constructors :		
DBCollection(Mongo mongo, String name)		
Method Summary:		
Modifier and Type	Method	Description
Void	addOption(int option)	Adds the give option
DBCollection	getCollection(String name)	Gets a collection with a given name.
Set<String>	getCollectionFromString(String s)	Gets a collection with a given name.

Mongo	getMongo()	Gets the Mongo instance
DBCollection	createCollection(String name, DBObjectoptions)	Creates a collection with a given name and options
Boolean	authenticate(String username, char[] password)	Authenticate database user with given username and password
void	dropDatabase()	Drop this database

2. DBCollection: (public abstract class DBCollection extends Object)

Package: com.mongodb.DBCollection

This class provides a skeleton implementation of a database collection

Constructors :		
DBCollection(DB base, String name)		
Method Summary:		
Modifier and Type	Method	Description
long	count()	returns the number of documents in this collection
void	createIndex(DBObjectkeys)	calls createIndex(com.mongodb.DBObject, com.mongodb.DBObject) with default index options
Void	drop()	Drops (deletes) this collection.
DBCursor	find(DBObjectref)	Queries for an object in this collection
WriteResult	insert(DBObject... arr)	Saves document(s) to the database
WriteResult	remove(DBObjecto)	calls remove(com.mongodb.DBObject, com.mongodb.WriteConcern) with the default WriteConcern

3. MongoClient: (public class MongoClientextends Mongo)

Package: com.mongodb.MongoClient

A MongoDB client with internal connection pooling. For most applications, you should have one MongoClient instance for the entire JVM.

Constructors :
MongoClient() : Creates an instance based on a (single) mongodb node (localhost, default port).
MongoClient(String host, int port): Creates a Mongo instance based on a (single) mongodb node. Parameters: host - the database's host address port - the port on which the database is running

4. BasicDBObject: (public class BasicDBObjectextends BasicBSONObject implements DBObject)

Package: com.mongodb.BasicDBObject

A basic implementation of bson objects that is mongo specific. A DBObject can be created as follows, using this class:

DBObjectobj = new BasicDBObject();

obj.put("foo", "bar");

Constructors :
public BasicDBObject(int size): creates an empty object
public BasicDBObject(int size): creates an empty object

5. DBCursor: (public class DBCursorextends Object implements Iterator<DBObject>, Iterable<DBObject>, Closeable)

Package: com.mongodb. DBCursor

An iterator over database results. Doing a find() query on a collection returns a DBCursor thus :

DBCursor cursor = collection.find(query);

if(cursor.hasNext())

DBObjectobj = cursor.next();

Constructors :		
public DBCursor(DBCollectioncollection,DBObjectq, DBObjectk, ReadPreferencepreference)): Initializes a new database cursor		
Method Summary:		
Modifier and Type	Method	Description
public boolean	hasNext()	Checks if there is another object available
public DBObject	Next()	Returns the object the cursor is at and moves the cursor ahead by one.
Public void	Remove()	Not implemented
Public int	Length()	pulls back all items into an array and returns the number of objects. Note: this can be resource intensive

- **MongoDB Overview:**

MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.

Database

Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

- **Collection**

Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

- **Document**

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

Below given table shows the relationship of RDBMS terminology with MongoDB

RDBMS	**MongoDB**
Database	Database
Table	Collection
Tuple/Row	Document
Column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by mongodb itself)
RDBMS	MongoDB

Sample document

Below given example shows the document structure of a blog site which is simply a comma separated key value pair

```
{
_id: ObjectId(7df78ad8902c)
title: 'MongoDB Overview',
description: 'MongoDB is no sql database',
by: 'tutorials point',
url: 'http://www.tutorialspoint.com',
tags: ['mongodb', 'database', 'NoSQL'],
likes: 100,
}
```

_id is a 12 bytes hexadecimal number which assures the uniqueness of every document. You can provide _id while inserting the document. If you didn't provide then MongoDB provide a unique id for every document. These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id, next 2 bytes for process id of mongodb server and remaining 3 bytes are simple incremental value.

Conclusion:

Here we have studied MongoDB Installation, Basic CRUD operations and Execution.

Questions:

1. Explain the concept of NoSQL?
2. Explain different types of Database?
3. What is the difference between SQL and NOSQL?
4. What is the need of NOSQL?
5. What are the CURD Operation explain with example?