# MALAVIYA NATIONAL INSTITUTE OF TECHNOLOGY JAIPUR

**Autumn Semester 2017-2018**

**Lab Sheet :** ***DATA MANUPULATION LANGUAGE***

---

**Objectives:**

- *Getting more familiar with DDL*
    - *DROP, DELETE ,TRUNCATE, ALTER TABLE*
    - *Ensuring data integrity through update/delete*
- *What is DML?*
    - *Brief overview of DML*
- *Getting familiar with DML*
    - *INSERT statement*
    - *SELECT statement and its various uses.*
        - *SELECT **
        - *SELECT using DISTINCT*

      c) Expressions and Strings in the SELECT Command

      *d) SELECT using 'LIKE' perator(pattern matching)*
- *SELECT using 'BETWEEN' operator.*
- *SELECT using SET Operators:*
    - *AND*
    - *OR*
    - *UNION*
    - *INTERSECT*
    - *MINUS*
    - *EXCEPT*
    - *IN*
    - *EXISTS*
    - *op ANY*
    - *op ALL*
- *SELECT using AGGREGATE FUNCTIONS:*
- *SELECT using ORDER BY, GROUP BY, and HAVING*
- *SELECT USING IS NULL and IS NOT NULL*
    - ***UPDATE statement***
    - ***DELETE statement***

---

- **How to delete /remove some specified selected number of rows from a table [that satisfying the condition specified in** *WHERE clause***]**

  **Syntax:** DELETE FROM *tableName* WHERE *Search-Condition;*

- **How to delete /remove all rows of a table :**

  **Syntax:** DELETE FROM *tableName;*

**Note:** *DELETE is a DML command not the DDL command.*
After performing a DELETE operation we can also roll back (using ROLLBACK)to undo change, if we didn't used COMMIT to make the change to be permanent. This also makes all associated DELETE triggers of the table to fire.

**Task 1:** To Delete from supplier table where id <400;

```
        SELECT * FROM supplier; //observer the contents of table supplier
Now run ,
        DELETE FROM TABLE supplier WHERE id<400;
SELECT * FROM supplier; // observe the effect on contents of the table supplier
```

- **How to TRUNCATE table:**
  What if we only want to delete the data inside the table, and not the table itself? Then, use the TRUNCATE TABLE statement. TRUNCATE TABLE command deletes all the rows in the table permanently without disturbing the integrity constraints and the data fields. This operation cannot be rolled back and no triggers will be fired.

  **Syntax:**

  **TRUNCATE TABLE tableName;**

  Try truncating the STUDENT table which you may create.

- **How to remove a table from databse:**

  **Syntax:** DROP TABLE tableName; //***Be careful as it remove table permanently from database***.

**Note:** With DROP command, all the tables' rows, indexes and privileges will also be removed. No DML triggers will be fired and this operation cannot be rolled back.

- **How to DROP Integrity constraints**

  ALTER TABLE *tableNname* DROP CONSTRAINT *constraint_name*;

- **How to ALTER table:**
  The ALTER TABLE statement can also be used to add, delete, or modify columns in an existingtable.
  We have seen the use of ALTER command while altering the constraints.
  Let's see some more uses of ALTER command:

  a) To add a column in a table, use the followingsyntax:
     ALTERTABLE tableName ADD (NewColumnName DataType);

  b) To delete columns in a table, use the following syntax:
     ALTER TABLE tableName DROP (Column name1),DROP(Column name2);

  c) To change the data type of a existing column in a table, using the syntax:
     ALTER TABLE tableName MODIFY (ColumnName NewDataType);
     OR
     ALTERTABLE tableName ALTER COLUMN ColumnName NewData Type;

Task 2: a) **Add a new column in supplier table and increase the size of name from 20 char to 30 char longer. //supplier table created in *Lab#2.***

b) **Drop NOT NULL & FOREIGN KEY constraint from product table created in lab#2**

c) **Use TRUNCATE command to remove all rows of product table.**
Thereafter that run SELECT * FROM product; // to observe the effect

d) **Drop the product table**
Now run SELECT * FROM product; // to observe theeffect

## Ensuring other issues of data integrity through update and delete

A foreign key data constraint specifies that values in one table must also appear in another table. For example **sid** column of ***product*** table (k/a child table) refers primary key column **id** of ***supplier*** table (k/a parent table) as foreign key as described above; which implies that column **sid** of ***product*** table takes only those values which must be appeared/exists in ***supplier*** table as values in **id** column .

The SQL REFERENCES clause prevents a foreign key value from being added if it is not already a valid value in the referenced primary key column, but there are other integrity issues too.
What about if **id** value is changed/deleted, the connection (relationship) between that ***supplier*** and ***product*** supplied by that supplier will be ruined. The REFERENCES clause prevents making such a change in the foreign key value, but not in the primary key value.
Consider populated supplier and product table records created above. If you delete supplier table entries, database will give error when there is an entry exists in product table which is referencing that particular id in supplier table which is a primary key.

This problem can be solved through the following:

i) Ensuring that primary key values cannot be changed once they are established.
**Using ON UPDATE RESTRICT option.**

ii) Pass the changes through to the child table(s).It means that if primary key value of referenced parent table changes, that change would also happen to the assocaited foreign key value in child table automatically.
**Using ON UPDATE CASCADE option.**

iii) Allow the update on primary key value of referenced parent table but change the involved foreign key value of child table that matches to NULL.

But this would result in losing the connection between the tables which is not a desired effect.

**Using ON UPDATE SET NULL option.**

iv) Allow the update on primary key value of referenced parent table but set/change the involved foreign key value of child table that matches to a predefined default value.

**Using ON UPDATE SET DEFAULT option.**

A foreign key with these additional constraints can be defined in either a CREATE TABLE statement or an ALTER TABLE statement.
We can apply the above described additional integrity constraints using the following syntax associated with updates as:

***a) Adding 'ON UPDATE CASCADE' option constraint while creating the table***

```
CREATE TABLE  TableName1
        (  Column   name1   Data   Type,
           Column   name2   Data   Type,
           Column name3       Data Type,
        CONSTRAINT constraint_name PRIMARY KEY (Column name1)
      );

CREATE TABLE  TableName2
                  (  Column   name4   Data   Type,
                     Column   name5   Data   Type,
                     Column name6       Data Type,
            CONSTRAINT constraint_name PRIMARY KEY (Column name4)
            CONSTRAINT constraint_name FOREIGN KEY (Column name4)
                        REFERENCES TableName1 (Column name1)
             ON UPDATE CASCADE
            );
```

**Note:** **In a table only a singleton constraint can be applied from CASCADE, RESTRICT, SET NULL, SET DEFAULT. Hence to apply any others two use RESTRICT, SET NULL or SET DEFAULT in place of CASCADE in the above syntax.**

***b) Adding 'ON UPDATE CASCADE' option constraint after creating the table***

In case you have not applied any foreign key constraint while creating the table as in previous example (in TableName2), that can be done using ALTER command:

```
ALTER TABLE TableName2
        ADD CONSTRAINT constraint_name FOREIGN KEY (Column name4)
         REFERENCES  TableName1  (Column name1)
          ON UPDATE CASCADE;
```

**Note:** **In a table only a singleton constraint can be applied from CASCADE, RESTRICT, SET NULL,SET DEFAULT .Hence to apply any others two use RESTRICT, SET NULL or SET DEFAULT in place of CASCADE in the above syntax.**

The above mentioned problems can also arise if **referenced primary key value is be deleted** which is referenced as foreign key in another table. For example if **id is** deleted from ***supplier*** table which is referenced by ***product*** table as foreign key via **sid** column, Then the relationship between **supplier** and ***product*** *table* will be ruined

This problem due to the deletion of referenced primary key will be also handled similar way as above by using ON DELETE instead of ON UPDATE:

ON DELETE CASCADE, ON DELETE RESTRICT, ON DELETE SET NULL and ON DELETE SET DEFAULT option.

**Task 3:** Create again the table **product (pid,sid,Pname)** such that this table assigns foreign key as **sid** that references the column ***id*** of the **supplier** table in case product table not available/exist.

Insert following records if not exist in supplier table:

```
INSERT INTO supplier (id,name) VALUES ( 220, 'Sundram');
INSERT INTO supplier VALUES ( 5555,'Sansar',6767676767676);
INSERT INTO supplier (id,name,phone) VALUES (434,'Krish',303030303030);
```

Insert these records if not exist in product table:

```
INSERT INTO product (pid,sid,Pname) VALUES (2349, 220, 'Laptop');
INSERT INTO product (pid,sid,Pname) VALUES (3449,5555, 'Mobile');
INSERT INTO product (pid,sid,Pname) VALUES (5489,434, 'Mobile');
```

Perform the following and observer the effect:

```
DELETE from supplier where id=5555;   //observe whether record removed
```

**Task 4: a)** Run the command: DROP TABLE product; //observe effect

**b)** Run the command to create product table:

```
CREATE TABLE product
 (  pid  int(4)   not null,
    sid int(4) not null,
 Pname  varchar(30)
);
```

**c)** Now since you forgot to apply referential constraint with ON DELETE option on product table created as:

```
ALTER TABLE product ADD CONSTRAINT fk_sup
         FOREIGN KEY (sid) REFERENCES supplier(id)
            ON DELETE CASCADE;
```

Now Insert these records again if not exist in product table:

```
INSERT INTO product (pid,sid,Pname) VALUES (2349, 220, ‘Laptop’);
INSERT INTO product (pid,sid,Pname) VALUES (3449,5555, ‘Mobile’);
INSERT INTO product (pid,sid,Pname) VALUES (5489,434, ‘Mobile’);
```

Once again perform the following and compare with the observation of task 8:

```
DELETE from supplier where id=5555;    //observe whether record removed or not
```

**<u>Data Manipulation Language (DML)</u>** –

Those command that are used to maintain and query a database. DML is used for inserting, updating, deleting and querying (retrieving) the data in the database. They may be issued interactively, so that a result is returned immediately following the execution of the statement or they may be included within programs written in a procedural programming language such as C, Java etc.

- **DML Commands** which are used to manipulate or retrieve the database are:

  a) **INSERT**: used to populate data into a table. It allows us to insert single or multiple records into database.

  b) **SELECT** - retrieve data from the database.

  c) **UPDATE** - updates existing data within a table.

  d) **DELETE** - deletes all records from a table, the space for the records remain

- **<u>INSERT statement</u>:** used to insert data into a table

**Syntax:** There are two variations:

***a) To insert single row of data***

**i)** INSERT INTO *TableName* (*ColumnName1,ColumnName2 ,….., ColumnNameN)*
VALUES (*values1, values2,…….., valuesN*)**;**

**ii)** INSERT INTO *TableName* VALUES ( *values1,values2,…..,valuesN*);

**<u>Caution</u>:** Whenever we insert data values this way, we must ensure to give the field values in proper order, i.e. order in which they have created in the table.

***b) To insert multiples records quickly from another table having same structure***

**i)** INSERT INTO *TableNname* **(***ColumnName1, ColumnName2, ColumnName3,…***)**
SELECT *ColumnName1, ColumnName2, ColumnName3,……*
FROM *TableName2* WHERE *Conditional-Expression ;*

**ii)** INSERT INTO *TableName*
SELECT *ColumnName1, ColumnName2, ColumnName3,……*
FROM *TableName2* WHERE *Conditional-Expression ;*

- **SELECT statement:** To extract i.e. retrieve records from one or more tables in the database.

  **Syntax:** **SELECT** [ DISTINCT | ALL ] column_lists
  [ **FROM** Tables_Name ]
  [ **WHERE** search_condition ]
  [ **GROUP BY** group_by_expression]
  [ **HAVING** search_condition]
  [ **ORDER BY** order_column_expr1, order_column_expr2, .... ] [ASC | DESC];

  - **SELECT statement and its various uses:**
    - **SELECT:** List the columns (and expressions) that should be returned from the query
    - **FROM:** Indicate the table(s) or view(s) from which data will be obtained
    - **WHERE:** Indicate the conditions under which a row will be included in the result
    - **GROUP BY:** Indicate categorization of results. Used to divide a table into subsets(by groups).It useful when paired with aggregate functions such as SUM,COUNT,AVG etc. and these aggregate function used to provide summary information for that group.
    - **HAVING:** Indicate the conditions under which a category (group) will be included that meets a criterion, rather than rows.
    - **ORDER BY:** Sorts the rows of result according to specified criteria

- **SELECT * : This statement displays all the data in the given table.**

  **Example:** SELECT * FROM SAILORS;

- **SELECT using DISTINCT:** The use of DISTINCT command in SELECT statement eliminates the duplicates in that column.

  **Example1:** SELECT ALL SNAME FROM SAILORS;
  SELECT SNAME FROM SAILORS;
  SELECT DISTINCT SNAME FROM SAILORS;
  //observer the effect & compare the output with the content of SAILORS table.

  **Example2:** To find the name and age of all sailors.

  SELECT DISTINCT S.SNAME , S.age AS Sailor_Age FROM SAILORS AS S;

  // distinguish between syntax of both example & Observer the output columns carefully

Note: Here **AS clause** used to specify an alias name for column(s) [or table(s)] in a SELECT statement [or WHERE statement]. Specified alias name will be appeared as column heading of the query result instead of the actual column name.

**Task 3:** Find all sailors with a rating above 7.

- **SELECT using LOGICAL Operators:** Logical operators used in the SQL sentences are: AND , OR and NOT.

**Example 3:** Find names from sailors who have reserved boat number 103.

```
SELECT S.sname FROM SAILORS S, RESERVES R
                WHERE S.sid=R.sid AND R.bid=103;
         //Observer the output;
```

**Now split this query in steps to observer how conceptually it executed:**

```
SELECT * FROM SAILORS S, RESERVES R;
    //observer its outcome, it generated a cross-product of the two tables.
```

**Now apply where clause to it as ,**

```
SELECT  * FROM SAILORS S, RESERVES R
                WHERE S.sid=R.sid AND R.bid=103;
      //observer its outcome with the above query
```

**Since we want sailors name only hence to display it use sname filed in select clause:**

```
SELECT S.sname FROM SAILORS S, RESERVES R
                WHERE S.sid=R.sid AND R.bid=103;
```

Finally this query lists name of sailors who have reserved boat number 103.

**Task 4:** Find the sids of sailors who have reserved a red boat.

**Example 4:** Find the names of sailors who have reserved a red or a green boat.

```
SELECT S.sname   FROM SAILORS  S, RESERVES  R, BOATS  B
                WHERE S.sid=R.sid AND R.bid=B.bid
                      AND  ( B.color ='red'  OR B.color ='green');
```

- **Expressions and Strings in the SELECT Command:** SQL supports a more general version of the select list than just a list of columns. Each item in a select list can be of the form **expression AS columnName** where expression is any arithmetic or string expression over columnName.

```
SELECT  sname,sid-10, rating*1.5,  age/2
        FROM SAILORS;
 //observer the output and its column name
```

**Example:** Compute increments for the ratings of persons who have sailed two different boats on the same day?

```
SELECT  S.sname,  S.rating+1  AS  rating
FROM  SAILORS S, RESERVES R1,RESERVES R2
WHERE S.sid=R1.sid AND S.sid=R2.sid
        AND R1.day =R2.day AND R1.bid<> R2.bid;
```

- **SELECT using LIKE Operator:** SQL supports **pattern** matching using the **LIKE** operator, along with the use of wild card symbols **%** (which stands for zero or more arbitrary characters match) and **_** (which stands for exactly one arbitrary character match).

For example, '_AB%' denotes a pattern which matching every string that contains at least three characters, with the 2nd and 3rd letter being 'A' and 'B' respectively.

```
SELECT bname, color
        FROM boats
        WHERE bname LIKE 'In%';
        //observer the output with boats table
```

**Example :** Find the ages of sailors whose name begin and end with B and has at least 3 characters?

```
SELECT S.age
FROM SAILORS S
WHERE S.sname LIKE 'B_%b';
```

- **SELECT using BETWEEN Operator:** BETWEEN operator is used in a WHERE clause to select a range of data between two values.

**Syntax:**
```
SELECT ColumnNname(s) FROM TableName
          WHERE ColumnName
                    BETWEEN value1 AND value2;
```

**Example:**
```
SELECT *     FROM sailors
                  WHERE rating BETWEEN 7 AND 10;
```

Now use NOT BETWEEN in above query.

<u>**Note:**</u> instead of above, comparison operator < and > can also use to establish range of values.

**Example:**
```
SELECT sid, sname   FROM sailors
            WHERE sname  NOT BETWEEN 'Zorba' AND 'Lubber';
```

- **SELECT using DISTINCT:** The use of DISTINCT command in SELECT statement eliminates the duplicates in that column.

**Example:** To find the name and age of all sailors.

```
SELECT DISTINCT S.SNAME  , S.age AS  Sailor_Age   FROM SAILORS AS  S;
///Observe The output
```

<u>**Note:**</u> Here **AS clause** used to specify an alias name for column(s) [or table(s)] in a SELECT statement [or WHERE statement]. Specified alias name will be appeared as column heading of the query result instead of the actual column name.

- **SELECT using SET Operators:** SQL provides SET manipulation constructs that extend the basic query form presented in previous lab sheet.
  SQL set operators allow to combine the results from two or more SELECT statements. Answer to these queries is a multiset of rows.

  Set operator types are:

**i) UNION** [DISTINCT] **ii) UNION ALL** **iii) EXCEPT** [DISTINCT] **iv) EXCEPT ALL**

**v)INTERSECT** [DISTINCT**]** **vi) INTERSECT ALL**

**N OTE:** oracle support MINUS instead of EXCEPT.

***Here,*** **ALL** clause: Combines the results of two SELECT statements into one result set and
**DISTICNT** clause: combines the results of two SELECT statements into one result set, and then eliminates any duplicate rows from that result set.

- **UNION**
  **Example :** Find the names of sailors who have reserved a red or a green boat.

  Above query which you are written using 'OR' , can be performed using 'UNION' perator**:**

```
SELECT S.sname
    FROM SAILORS S, RESERVES R,BOATS B
    WHERE S.sid=R.sid AND R.bid=B.bid AND  B.color ='red'
    UNION
    SELECT S2.sname
    FROM SAILORS S2, RESERVES R2,BOATS B2
    WHERE S2.sid=R2.sid AND R2.bid=B2.bid AND  B2.color ='green';
```

- **INTERSECT**
  **Example:** Find names of the sailors who have reserved both 'red' and 'green' boats?

```
SELECT S.sname  FROM SAILORS S,RESERVES R, BOATS B
WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='red'
INTERSECT
SELECT S2.sname  FROM SAILORS S2,RESERVES R2, BOATS B2
WHERE S2.sid=R2.sid AND R2.bid=B2.bid AND B2.color='green';
```

//This query can also be written using 'AND' operator. Try using AND operators only.

- **MINUS or EXCEPT**
  **Example:** Find sids of all sailors who have reserved 'red' boats but not 'green' boats?

```
SELECT R.sid
FROM RESERVES R, BOATS B
WHERE S R.bid=B.bid AND B.color='red'
MINUS
SELECT R2.sid
FROM RESERVES R2, BOATS B2
WHERE R2.bid=B2.bid AND B2.color='green';
```

Task 1: Write the above query of examples using BOATS, SAILORS and RESERVES table through EXCEPT operator.
Task 2: Find sids of all sailors who have rating of 10 or reserved boat 104 using UNION.

- **Independent Nested Queries:** One of the most powerful features of SQL is nested queries. A nested query is a query that has another query embedded within it; the embedded query is called a sub query. A sub query typically appears within the WHERE clause of a query. Sometimes it appears in the FROM clause or 'HAVING' clause.
  - **IN and NOT IN with Lists**
    - **IN:** To match a list of values. Useful in sub queries.

**Syntax:**
```
SELECT  ColumnName(s) FROM   TableName(s)
               WHERE  ColumnName  IN (value1,value2,.....valueN);
```

The list (set of values) inside the parentheses after IN can be literals or can be a SELECT statement with a single result column, the result of which can will be plugged in as the set of values for comparison.

**Example :** Find the names of sailors who have reserved boat 103?

```
SELECT S.sname
FROM SAILORS S
WHERE S.sid IN (SELECT R.sid
                FROM RESERVES R
                WHERE R.bid=103);
```

**NOTE:** It's easier to find out all sailors who have not reserved boat 103, just replace IN by **NOT IN**.

**Example :** Find the names of sailors who have reserved a red boat?

```
SELECT    S.sname
FROM   SAILORS   S
WHERE    S.sid    IN
(SELECT R.sid
                FROM RESERVES R
                WHERE R.bid IN (SELECT B.bid FROM BOATS B
                                WHERE B.color='red') );
```

Task: Find the names of sailors who have not reserved a red boat?

```
SELECT S.sname
FROM SAILORS S
WHERE S.sid NOT IN (SELECT R.sid
                FROM RESERVES R
                WHERE R.bid IN (SELECT B.bid FROM BOATS B
                                WHERE B.color='red') );
```

- **Correlated Nested Queries:** In these queries the inner subquery could depend on the row currently being examined by the outer sub query.
  - **EXISTS:**

**Example 10:** Find the name of the sailors who have reserved boat 103?

```
SELECT S.sname
FROM SAILORS S
  WHERE EXISTS (  SELECT *
                          FROM RESERVES R
                          WHERE R.bid=103  AND R.sid=S.sid
                 );
```

**NOTE:** By using NOT EXISTS instead of EXISTS we can compute the names of sailors who have not reserved a 'red' boat.

- **SQ L also supports op ANY and op ALL ,where op is one of the arithmetic comparison operators { <,<=,=,<>,>,>=,>}**
  - **op ANY**

    **Example :** Find sailors whose rating is better than some sailor called 'horatio'?

```
SELECT    S.sid
    FROM SAILORS S
         WHERE S.rating >ANY (SELECT s2.rating
                              FROM SAILORS s2
                              WHERE s2.sname='Horatio');
```

- **op ALL**

  **Example :**Find sailors whose rating is better than every sailor called 'Horatio'? **SELECT** S.sid

```
FROM SAILORS S
WHERE S.rating >ALL (SELECT s2.rating
                              FROM SAILORS s2
                              WHERE s2.sname='Horatio');
```

**NOTE**: IN and NOT IN are equivalent to =ANY and <> ALL respectively.

<u>Task</u> **:** Find the sailors with the highest rating? (Hint: use >=ALL operator)

**Example:** Compute increments for the ratings of persons who have sailed two different boats on the same day?

```
SELECT  S.sname,  S.rating+1  AS  rating
FROM  SAILORS S, RESERVES R1,RESERVES R2
WHERE S.sid=R1.sid AND S.sid=R2.sid
          AND R1.day =R2.day AND R1.bid<> R2.bid;
```

<u>Task</u> **:** Find sailors whose rating range is between 7 and 10 using comparison operator.

- **SELECT using AGGREGATE FUNCTIONS:**

  SQL allow some powerful class of constructs for computing aggregate values.

These functions used in SELECT list and in ORDER BY and HAVING clauses. But these are commonly used with GROUP BY clause in SELECT statement. Most frequently used functions are:

1. **COUNT([DISTINCT] X)**: returns the number of (unique) values(rows) in the X column.
2. **COUNT(*):** returns number of rows In the table including duplicates and those with Null values
3. **AVG([DISTINCT] X) :** The average of all (unique) values in the X column.
4. **SUM([DISTINCT] X):** The sum of all (unique) values in the X column.
5. **MAX(X):** The maximum value in the X column
6. **MIN(X):** The minimum value in the X column
7. **LOWER(char):**returns with all letters in lowercase
8. **INITCAP(char):**returns string with the first letter in upper case

**There are many more aggregate functions in MySQL apart from those mentioned above. Refer MySQL help/manual for other aggregate functions.**

Note: null values ignored by SUM, AVG, COUNT, MAX, and MIN but doesn't by COUNT(*).
Optional keyword DISTINCT can be used with SUM, AVG, and COUNT to eliminate duplicate values before an aggregate function is applied but the default is ALL.

**Example:** Find the average age of all sailors.

```
SELECT AVG(age)  "Average Age"  FROM SAILORS;
```

**Example:** Find the maximum and minimum ratings of sailors.

```
SELECT MAX(rating), MIN(rating)      FROM SAILORS;
```

**Example:** Count the number of different sailor's names.

**SELECT COUNT(DISTINCT S. sname)  FROM** SAILORS S;

**Now run** : SELECT COUNT(*)  FROM SAILORS S;  //observer the output with previous query.

- **SELECT using ORDER BY, GROUP BY, and HAVING clause:** So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we like to apply them to each of several *groups* of tuples.

  **Consider For example :** *Find the age of the youngest sailor for each rating level.*

  - In general, we don't know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (?):

```
SELECT MIN(S.age)
       FROM SAILORS S
            WHERE S.rating =i;
       Where  i=1,2,...10
```

Writing 10 such queries is tedious. More important we may not know what rating levels exist in advance.
To write such queries, we need a major extension to the basic SQL query form, namely the GROUP BY clause. This extension also includes an optional 'HAVING' clause that can be used to specify qualifications over groups.

**The general form of an SQL query with these extensions is:**

```
SELECT [DISTINCT] select-list
       FROM  from-list
            WHERE   qualification
                 GROUP BY grouping-list
                          HAVING  group-qualification;
```

So above query can be rewritten as:

```
SELECT S.rating, MIN(S.age)
    FROM SAILORS S
           GROUP BY S.rating;
```

**Example :** Find the average age of sailors for each rating level that has at least two sailors?

```
SELECT S.rating ,AVG(S.age) AS aveage
        FROM SAILORS S
                GROUP BY S.rating
                        HAVING COUNT(*)>1;
```

**Note:** HAVING clause can have a nested sub query.

Above Query in Example can be written using nested HAVING as :

```
SELECT S.rating , AVG(S.age) AS avgAge
    FROM SAILORS S
          GROUP BY S.rating
      HAVING 1< (SELECT COUNT (*) FROM SAILORS S2 WHERE S.rating =S2.rating);
```

**Example :** Find age of youngest sailor who is eligible to vote for each rating level with at least two such sailors.

```
SELECT S.rating ,MIN(S.age) AS MinAge
        FROM SAILORS S WHERE S.age>=18
                GROUP BY S.rating
                        HAVING COUNT(*)>1;
```

**Example:** Run the following query and observe output to the previous query output.

```
SELECT S.rating ,MIN(S.age) AS MinAge
```

```
FROM SAILORS S WHERE S.age>=18
        GROUP BY S.rating
                HAVING COUNT(*)>1 AND EVERY (S.age<=60);
```

**ORDER BY:**

- The ORDER BY keyword is used to sort the result-set by a specified column.
- The ORDER BY keyword sorts the records in ascending order by default.
- If you want to sort the records in a descending order, you can use the DESC keyword.

**Syntax form of ORDER BY:**

```
SELECT  column_name(s)
        FROM table_name
                        ORDER BY column_name(s) ASC|DESC;
```

**Example :** sort sailors name in ascending and descending order?

**In ascending order:**

```
SELECT  S.sname
        FROM   SAILORS S
                GROUP BY S.sname ASC;
```

**In descending order:**

```
SELECT S.sname
        FROM   SAILORS S
                GROUP BY S.sname DESC
```

- **SELECT using IS NULL and IS NOT NULL :** *How to test for NULL values?*
  – It is not possible to test for NULL values with comparison operators, such as =, <, or <>.
  – We will have to use the **IS NULL** and **IS NOT NULL** operators instead.

**Example :**

a) **IS NULL:**

```
SELECT S.sid
        FROM SAILORS S
              WHERE S.sname IS NULL;
```

b) **IS NOT NULL:**

```
SELECT  S.sid
                FROM SAILORS S
                WHERE S.sname IS NOT NULL;
```

**Note:** May your table doesn’t contain NULL values hence to practice create /use existing table with few null values inserted in them.

**UPDATE Statement**: It is used to update existing records in a table. It updates a single record or multiple records in a table.

Syntax: UPDATE table_name

```
SET column1=value, column2=value2, .....
WHERE expr_condition;
```

Example :

```
UPDATE  SAILORS
SET AGE=45      WHERE  sid=22;
```

- **DELETE Statement:** It is used to delete rows in a table. It allows to delete a single record or multiple records from a table.

Syntax: DELETE FROM table_name WHERE expr_condition;

Example:

```
DELETE  FROM RESERVES R
        WHERE R.bid=101 AND R.sid=22;
```

***Exercise:***

1) Find the names of sailors who have reserved a green boat?
2) Find the color of the boat reserved by 'Lubber'?
3) Find the ages of sailors whose name begins and ends with B and has at least three characters.
4) Find all sid's of sailors who have a rating of 10 or reserved boat 104?
5) Find the name of the sailors who have reserved a 'red' boat?( using IN)
6) Find the names of sailors who have not reserved a 'red' boat?(using NOT IN)
7) Find the names of sailors who have reserved both a 'red' and a 'green' boat?
8) Find the increments for the ratings of sailor who have sailed two different boats on the same day.
9) Find the name of sailors such that it have following relationship between their ratings:

2 x rating( 1st sailor) = rating (2nd sailor) - 1