

# **Parallel and Distributed Computing**



## **Lab Manual**

**By**

**Maham Arooj 2019-GCUF-065675**

**BSCS**

**Department of Computer Science**

**Government College University Faisalabad**



## **TABLE OF CONTENTS**

<b>EXPT. NO.</b>	<b>EXPERIMENT NAME</b>	<b>PAGE NO.</b>
1	Distributed Database for Bookstore	
2	Deadlock Detection Algorithm for distributed database using wait- for graph	
3	Object Oriented Database - Extended Entity Relationship (EER) model for University Database	
4	Parallel Database - University Counselling for Engineering colleges	
5	Parallel Database - Implementation of Parallel Join & Parallel Sort Algorithm	
6	Active Database - Implementation of Triggers & Assertions for Bank Database	
7	Deductive Database - Constructing Knowledge Database for Kinship Domain (Family Relations)	
8	Study and Working of WEKA Tool	
9	Query Processing - Implementation of an Efficient Query Optimizer	
10	Designing XML Schema for Company Database	

## **BASIC COMMANDZ -- USEFUL**

### **FOR LAB**

#### **SQL – STRUCTURED QUERY LANGUAGE**

##### **Tables**

In relational database systems (DBS) data are represented using tables (relations).

A query issued against the DBS also results in a table.

A table has the following structure:

Column 1	Column 2	...	Column n
			← Tuple
...	...	...	...

A table is uniquely identified by its name and consists of rows that contain the stored information, each row containing exactly one tuple (or record). A table can have one or more columns.

A column is made up of a column name and a data type, and it describes an attribute of the tuples. The structure of a table, also called relation schema, thus is defined by its attributes.

The type of information to be stored in a table is defined by the data types of the attributes at table creation time.

SQL uses the terms table, row, and column for relation, tuple, and attribute, respectively.

##### **Oracle offers the following basic data types:**

- **char(n):** Fixed-length character data (string), n characters long. The maximum size for n is 255 bytes (2000 in Oracle8). Note that a string of type char is always padded on right with blanks to full length of n. (+ can be memory consuming).

Example: char(40)

- **varchar2(n):** Variable-length character string. The maximum size for n is 2000 (4000 in Oracle8). Only the bytes used for a string require storage.

Example: varchar2(80)

- **number(o, d):** Numeric data type for integers and reals. o = overall number of digits, d = number of digits to the right of the decimal point.

Maximum values: o = 38, d = -84 to +127. Examples: number(8), number(5,2)

Note that, e.g., number(5,2) cannot contain anything larger than 999.99 without resulting in an error. Data types derived from number are int[eger], dec[imal], smallint and real.

- **date:** Date data type for storing date and time.

The default format for a date is: DD-MMM-YY. Examples: '13-OCT-94', '07-JAN-98'

- **long:** Character data up to a length of 2GB. Only one long column is allowed per table.

**Note:**

In Oracle-SQL there is no data type boolean. It can, however, be simulated by using either char(1) or number(1).

As long as no constraint restricts the possible values of an attribute, it may have the special value null (for unknown). This value is different from the number 0, and it is also different from the empty string "".

**Further properties of tables are:**

- The order in which tuples appear in a table is not relevant (unless a query requires an explicit sorting).
  - A table has no duplicate tuples (depending on the query, however, duplicate tuples can appear in the query result).

**STUDY- DATA DEFINITION IN SQL**

**Creating Tables**

The SQL command for creating an empty table has the following form:

```
create table <table> (
  <column 1> <data type> [not null] [unique] [<column constraint>],
  .....
  <column n> <data type> [not null] [unique] [<column constraint>],

  [<table constraint(s)>]

);
```

For each column, a name and a data type must be specified and the column name must be unique within the table definition. Column definitions are separated by comma. There is no difference between names in lower case letters and names in upper case letters. In fact, the only place where upper and lower case letters matter are strings comparisons.

A **not null** constraint is directly specified after the data type of the column and the constraint requires defined attribute values for that column, different from null.

Unless the condition not null is also specified for this column, the attribute value null is allowed and two tuples having the attribute value null for this column do not violate the constraint.

The keyword **unique** specifies that no two tuples can have the same attribute value for this column.

### **Checklist for Creating Tables**

The following provides a small checklist for the issues that need to be considered before creating a table.

- What are the attributes of the tuples to be stored? What are the data types of the attributes? Should varchar2 be used instead of char?
- Which columns build the primary key?
- Which columns do (not) allow null values? Which columns do (not) allow duplicates?
- Are there default values for certain columns that allow null values?

### **Modifying Table and Column Definitions**

It is possible to modify the structure of a table (the relation schema) even if rows have already been inserted into this table.

A column can be added using the alter table command

```
alter table <table>
```

```
add(<column> <data type> [default <value>] [<column constraint>]);
```

If more than only one column should be added at one time, respective add clauses need to be separated by colons. A table constraint can be added to a table using

```
alter table <table> add (<table constraint>);
```

**Note:**

A column constraint is a table constraint, too. not null and primary key constraints can only be added to a table if none of the specified columns contains a null value. Table definitions can be modified in an analogous way. This is useful, e.g., when the size of strings that can be stored needs to be increased. The syntax of the command for modifying a column is

```
alter table <table>
```

```
modify(<column> [<data type>] [default <value>] [<column constraint>]);
```

**Note:**

In earlier versions of Oracle it is not possible to delete single columns from a table definition. A workaround is to create a temporary table and to copy respective columns and rows into this new table.



### **Renaming a Table**

A table can be renamed using the rename command.

```
rename <old table name> to <new table name>;
```

### **Deleting a Table**

A table and its rows can be deleted by issuing the command

```
drop table <table> [cascade constraints];
```

## **STUDY - DATA MANIPULATION LANGUAGE (DML)**

After a table has been created using the create table command, tuples can be inserted into the table, or tuples can be deleted or modified.

### **Insertions**

The most simple way to insert a tuple into a table is to use the insert statement

```
insert into <table> [(<column i, . . . , column j>)]  
values (<value i, . . . , value j>;
```

For each of the listed columns, a corresponding (matching) value must be specified. Thus an insertion does not necessarily have to follow the order of the attributes as specified in the create table statement. If a column is omitted, the value null is inserted instead. If no column list is given, however, for each column as defined in the create table statement a value must be given.

If there are already some data in other tables, these data can be used for insertions into a new table. For this, we write a query whose result is a set of tuples to be inserted. Such an insert statement has the form

```
insert into <table> [(<column i, . . . , column j>)] <query>
```

### **Updates**

For modifying attribute values of (some) tuples in a table, we use the update statement:

```
update <table> set  
  
<column i> = <expression i>, . . . , <column j> = <expression j>  
  
[where <condition>;
```

An expression consists of either a constant (new value), an arithmetic or string operation, or an SQL query.

Note: that the new value to assign to <column i> must a matching data type.

An update statement without a where clause results in changing respective attributes of all tuples in the specified table. Typically, however, only a (small) portion of the table requires an update.

### **Deletions**

All or selected tuples can be deleted from a table using the delete command:

```
delete from <table> [where <condition>];
```

If the where clause is omitted, all tuples are deleted from the table. An alternative command for deleting all tuples from a table is the truncate table <table> command. However, in this case, the deletions cannot be undone. **Selections**

### **Selecting Columns**

The columns to be selected from a table are specified after the keyword select. This operation is also called ***projection***.

The query

```
select <column i, . . . , column j> from <table>;
```

lists only the attribute values of specified columns for each tuple from the denoted relation.

If all columns should be selected, the asterisk symbol “\*” can be used to denote all attributes.

The query

```
select * from <table>;
```

retrieves all tuples with all columns from the table.

Instead of an attribute name, the select clause may also contain arithmetic expressions involving arithmetic operators etc.

The query

```
Select <column l > * 1.55 from <table>;
```

retrieves all tuples with column specified as a product with the specified number from the table.

For the different data types supported in Oracle, several operators and functions are provided:

- for numbers: abs, cos, sin, exp, log, power, mod, sqrt, +, -, \_, /, . . .

- for strings: chr, concat(string1, string2), lower, upper, replace(string, search string, replacement string), translate, substr(string, m, n), length, to date, . . .
- for the date data type: add month, month between, next day, to char, . . .

Inserting the keyword **distinct** after the keyword select, duplicate result tuples are automatically eliminated.

```
Select distinct <column i> from <table>;
```

### Order by

It is also possible to specify a sorting order in which the result tuples of a query are displayed. For this the **order by clause** is used and which has one or more attributes listed in the select clause as parameter, **desc** specifies a descending order and **asc** specifies an ascending order (this is also the default order). The query

```
Select <column i, . . . , column j>
```

```
From <table>
```

```
Order by <column i> [asc], <column j> desc;
```

displays the result in an ascending order by the attribute <column i>. If two tuples have the same attribute value, the sorting criteria is a descending order by the attribute values of

<column j>.

### **Selection of Tuples**

#### **Where**

To conditionally select the datas from a table, we use the **where** keyword.

The syntax is as follows:

```
Select <column name>
From <table>
Where "condition"
```

#### **And Or**

the where keyword can be used to conditionally select data from a table. This condition can be a simple condition (like the one presented in the previous section), or it can be a compound condition. Compound conditions are made up of multiple simple conditions connected by **AND** or **OR**. There is no limit to the number of simple conditions that can be present in a single SQL statement.

The syntax for a compound condition is as follows:

```
Select "column_name"
From "table_name"
Where "simple condition"
{[AND|OR] "simple condition"}+
```

The {}+ means that the expression inside the bracket will occur one or more times. Note that **AND** and **OR** can be used interchangeably. In addition, we may use the parenthesis sign ( ) to indicate the order of the condition.

#### **IN Keyword**

The **IN** keyword, when used with where in this context, we know exactly the value of the returned values we want to see for at least one of the columns. The syntax for using the **IN** keyword is as follows:

```
Select <column name>
From <table>
Where <column name> IN ('value1', 'value2' ...)
```

The number of values in the parenthesis can be one or more, with each values separated by comma. Values can be numerical or characters. If there is only one value inside the parenthesis, this command is equivalent to

```
Select <column name>
From <table>
Where <column name> NOT IN ('value1', 'value2' ...)
```

### Between

Whereas the IN keyword help people to limit the selection criteria to one or more discrete values, the *BETWEEN* keyword allows for selecting a range.

The syntax for the BETWEEN clause is as follows:

```
Select <column name>
From <table>
Where <column name> BETWEEN 'value1' AND 'value2'
```

This will select all rows whose column has a value between 'value1' and 'value2'.

For all data types, the comparison operators =, != or <>, <, >, <=, >= are allowed in the conditions of a where clause.

For a tuple to be selected there must (not) exist a defined value for this column.

value: <column> is [not] null

*Note:* The operations = null and != null are not defined!

Domain conditions: <column> [not] between <lower bound> and <upper bound>

### Delete

To delete particular entries of table when satisfies a particular condition

The syntax for the Delete is as follows:

```
Delete from <table> where <condition>;
```

## **STUDY- CONSTRAINTS**

Constraints are used to limit the type of data that can go into a table. Such constraints can be specified when the table is first created via the **CREATE TABLE** statement, or after the table is already created via the **ALTER TABLE** statement.

Common types of constraints include the following:

- **NOT NULL Constraint**: Ensures that a column cannot have NULL value.
- **DEFAULT Constraint**: Provides a default value for a column when none is specified.
- **UNIQUE Constraint**: Ensures that all values in a column are different.
- **CHECK Constraint**: Makes sure that all values in a column satisfy certain criteria.
- **PRIMARY KEY Constraint**: Used to uniquely identify a row in the table.
- **FOREIGN KEY Constraint**: Used to ensure referential integrity of the data.

### **NOT NULL Constraint**

By default, a column can hold NULL. If you not want to allow NULL value in a column, you will want to place a constraint on this column specifying that NULL is now not an allowable value.

The syntax to be used is

```
Create table <table> (  
<column 1> <data type> [NOT NULL] [<column constraint>],  
.....  
<column n> <data type> [NOT NULL] [<column constraint>],  
);
```

### **DEFAULT Constraint**

The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value.

The syntax used is

```
Create table <table> ( <column  
1> <data type>,  
.....  
<column n> <data type>,  
<column n> <data type> DEFAULT value  
);
```

### **UNIQUE Constraint**

The UNIQUE constraint ensures that all values in a column are distinct.

The syntax used is

```
Create table <table> (  
<column 1> <data type> UNIQUE,  
.....  
<column n> <data type>  
);
```

### **CHECK Constraint**

The CHECK constraint ensures that all values in a column satisfy certain conditions. Once defined, the database will only insert a new row or update an existing row if the new value satisfies the CHECK constraint. The CHECK constraint is used to ensure data quality. The syntax used is

```
Create table <table> (  
<column 1> <data type> CHECK <condition>,  
.....  
<column n> <data type>  
);
```

### **PRIMARY KEY Constraint**

A primary key is used to uniquely identify each row in a table. It can either be part of the actual record itself, or it can be an artificial field (one that has nothing to do with the actual record). A primary key can consist of one or more fields on a table. When multiple fields are used as a primary key, they are called a composite key.

Primary keys can be specified either when the table is created (using **CREATE TABLE**) or by changing the existing table structure (using **ALTER TABLE**). The syntax used is

```
Create table <table> (  
<column 1> <data type> PRIMARY KEY,  
.....  
<column n> <data type>  
);
```

Or

```
Create table <table> ( <column  
1> <data type>,  
.....  
<column n> <data type>  
);  
Alter table Customer ADD PRIMARY KEY <column name>;
```

Note: Before using the ALTER TABLE command to add a primary key, you'll need to make sure that the field is defined as 'NOT NULL' -- in other words, NULL cannot be an accepted value for that field.

### **FOREIGN KEY Constraint**

A foreign key is a field (or fields) that points to the primary key of another table. The purpose of the foreign key is to ensure referential integrity of the data. In other words, only values that are supposed to appear in the database are permitted.

The syntax used is

```
Create table <table> (  
<column 1> <data type> PRIMARY KEY,  
.....  
<column n> <data type>  
);  
Create table <table1> (  
<column 1> <data type> PRIMARY KEY,  
.....  
<column m> <data type> REFERENCES <table>  
);
```

Or

```
Create table <table> (  
<column 1> <data type> PRIMARY KEY,  
.....  
<column n> <data type>  
);
```

```
Create table <table1> (  
<column 1> <data type> PRIMARY KEY,  
.....  
<column m> <data type>  
);
```

```
Alter table <table1>  
ADD (CONSTRAINT fk_<table1>) FOREIGN KEY <column m> REFERENCES <table>(<column n>);
```



## **STUDY- JOINS**

This is a binary operation that allows two relations to combine certain selections and cartesian product into one resulting relation.

### **TYPES OF JOIN**

- Inner join
- Outer join

### **INNER - JOIN**

Here, join operation forms a cartesian product of two relation's arguments, performs a selection forcing equality on those attributes that appear in both relation schemes and finally removes duplicate attributes. Also, this is referred as inner join.

The query

```
Select *  
From <table1>,<table2>  
Where table1.column i = table2.column i;
```

## **OUTER – JOIN**

The outer-join operation is an extension of the join operation to deal with missing information.

There are three forms of outer – join

- Left outer - join
- Right outer - join
- Full outer - join

### **Left Outer – Join**

This takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with null values for all other attributes from the right relation and adds them to the result of the join operation.

The query

```
Select *  
From <table1>,<table2>  
Where table1.column i(+) = table2.column i;
```

### **Right Outer – Join**

This takes all tuples in the right relation that did not match with any tuple in the left relation, pads the tuples with null values for all other attributes from the left relation and adds them to the result of the join operation.

```
Select *  
From <table1>,<table2>  
Where table1.column i = table2.column i(+);
```

### **Full Outer – Join**

This pads tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from the left relation and adds them to the resultant relation.

*Note:* The syntax for performing an outer join in SQL is database-dependent. In Oracle, we will place an "(+)" in the **WHERE** clause on the other side of the table for which we want to include all the rows.

## **STUDY-VIEWS**

A view is a virtual table. A view consists of rows and columns just like a table. The difference between a view and a table is that views are definitions built on top of other tables (or views), and do not hold data themselves. If data is changing in the underlying table, the same change is reflected in the view. A view can be built on top of a single table or multiple tables. It can also be built on top of another view.

Views are allowed to use one of the following constructs in the view definition:

- Joins
- Aggregate function such as sum, min, max etc.
- Set-valued subqueries (in, any, all) or test for existence (exists)  $\square\square$   
Group by clause or distinct clause

Views are classified as

- Read - Only views  $\square\square$  Updatable views

### **Read – Only Views**

These views are created from multiple relations.

### **Updatable Views**

These views are created from individual relation.

Views offer the following advantages:

1. **Ease of use:** A view hides the complexity of the database tables from end users. Essentially we can think of views as a layer of abstraction on top of the database tables.
2. **Space savings:** Views takes very little space to store, since they do not store actual data.
3. **Additional data security:** Views can include only certain columns in the table so that only the nonsensitive columns are included and exposed to the end user. In addition, some databases allow views to have different security settings, thus hiding sensitive data from prying eyes.

### **CREATE VIEW**

A view can be created using command create. The syntax

Create view <view name> AS <SQL Select Statement> (constraint<name>);
---

## **DELETE VIEW**

A view can be deleted using the command drop.

The syntax

Drop view <view-name>;
------------------------

## **STUDY- NESTED QUERIES**

A query result can also be used in a condition of a where clause. In such a case the query is called a subquery and the complete select statement is called a nested query.

A respective condition in the where clause then can have one of the following forms:

### 1. Set-valued subqueries

<expression> [not] in (<subquery>) <expression> <comparison operator> [any all] (<subquery>)
---

An <expression> can either be a column or a computed value.

### 2. Test for (non)existence

[not] exists (<subquery>)
---------------------------

In a where clause conditions using sub queries can be combined arbitrarily by using the logical connectives and and or.

Conditions of the form <expression> <comparison operator> [any|all] <subquery> are used to compare a given <expression> with each value selected by <subquery>.

- For the clause any, the condition evaluates to true if there exists at least one row selected by the subquery for which the comparison holds. If the subquery yields an empty result set, the condition is not satisfied.
- For the clause all, in contrast, the condition evaluates to true if for all rows selected by the subquery the comparison holds. In this case the condition evaluates to true if the subquery does not yield any row or value.

For all and any, the following equivalences hold:

in , = any not in , <>

all or != all

Often a query result depends on whether certain rows do (not) exist in (other) tables. Such type of queries is formulated using the exists operator.

## **STUDY- SET OPERATIONS**

Sometimes it is useful to combine query results from two or more queries into a single result.

SQL supports three set operators which have the pattern:

<query 1> <set operator> <query 2>

The set operators are:

- union [all] returns a table consisting of all rows either appearing in the result of <query1> or in the result of <query 2>.

Duplicates are automatically eliminated unless the clause all is used.

- intersect returns all rows that appear in both results <query 1> and <query 2>.
- minus returns those rows that appear in the result of <query 1> but not in the result of <query 2>.

## **STUDY- CURSORS**

A cursor is a SELECT statement that is defined within the *declaration* section of your PLSQL code. We'll take a look at three different syntaxes for cursors.

A cursor must be declared and opened before it can be used, and it must be closed to deactivate it after it is no longer required. Once the cursor has been opened, the rows of the query result can be retrieved one at a time using a FETCH statement.

### **DECLARE CURSOR**

The DECLARE CURSOR statement defines the specific SELECT to be performed and associates a cursor name with the query.

#### **Cursor without parameters**

The basic syntax for a cursor without parameters is:

```
CURSOR cursor_name  
IS  
SELECT_statement;
```

#### **Cursor with parameters**

The basic syntax for a cursor with parameters is:

```
CURSOR cursor_name (parameter_list)
IS
SELECT_statement;
```

### **OPEN CURSOR**

The OPEN statement executes the query and identifies all the rows that satisfy the query search condition, and positions the cursor before the first row of this result table.

Syntax

```
OPEN <cursor name>;
```

### **FETCH CURSOR**

The FETCH statement retrieves the next row of the active set.

Syntax

```
FETCH <cursor name>
INTO {host variable [indicator var i],[]}
```

### **CLOSE CURSOR**

The CLOSE statement is used to close the cursor that is currently open.

Syntax

```
CLOSE <cursor name>
```

## **STUDY- CONTROL STRUCTURES**

### **IF STATEMENTS**

The structure of the PL/SQL IF statement is similar to the structure of IF statements in other procedural languages. It allows PL/SQL to perform actions selectively based on conditions.

There are three forms of IF statements: ☐☐

#### **IF-THEN-END IF**

Syntax:

```
IF condition THEN  
Statements; END IF;
```

#### ☐☐ **IF-THEN-ELSE-END IF**

Syntax:

```
IF condition THEN  
    Statement 1;  
ELSE  
    Statement 2;  
END IF;
```

#### ☐☐ **IF-THEN-ELSIF-END IF**

Syntax:



```

IF condition THEN
    Statement 1;
[ELSIF condition THEN
    Statement 2;]

.....
[ELSIF
    Statements n;]
END IF;

```

If

**FOR** If the condition is FALSE or NULL, PL/SQL ignores the statements in the IF block. In either case, control moves to the next statement in the program following the END IF.

### **LOOP**

The

loops have the general structure as the basic loop. In addition they have a control statement before LOOP keyword to determine the number of iterations that PL/SQL performs.

syntax for the for loop is

```

FOR counter IN [REVERSE]
    Lower_bound.....upper_bound LOOP
    Statement 1;
    Statement 2;
END LOOP;

```

COUNTER - An explicitly declared integer whose value automatically increases by 1 on each iteration of the loop until the upper or lower bound is reached.

REVERSE - Causes the counter to decrement with each iteration from the upper bound to the lower bound.

LOWER\_BOUND - The lower bound for the range of counter values.

UPPER\_BOUND - The upper bound for the range of counter values.

### **WHILE LOOP**

The WHILE loop is used to repeat a sequence of statements until the controlling condition is no longer true. The condition is evaluated at the start of each iteration. The loop terminates when the condition is false. If the condition is false at the start of the loop, then no further iterations are performed.

The syntax for while loop is

```
WHILE condition LOOP Statement
    1;
    Statement 2; END
LOOP;
```

### **STUDY- PROCEDURES**

The syntax for a procedure is:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
    [ (parameter [,parameter]) ]
IS
    [declaration_section]
BEGIN
    executable_section
[EXCEPTION
    exception_section]
END [procedure_name];
```

When you create a procedure or function, you may define parameters. There are three types of parameters that can be declared:

1. **IN** - The parameter can be referenced by the procedure or function. The value of the parameter can not be overwritten by the procedure or function.
2. **OUT** - The parameter can not be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **IN OUT** - The parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

The optional clause or replace re-creates the procedure. A procedure can be deleted using the command drop procedure <procedure name> .

## **STUDY- FUNCTIONS**

The syntax for a function is:

```
CREATE [OR REPLACE] FUNCTION function_name
  [ (parameter [,parameter]) ]
  RETURN return_datatype
IS | AS
  [declaration_section]
BEGIN
  executable_section
[EXCEPTION
  exception_section]
END [function_name];
```

When you create a procedure or function, you may define parameters. There are three types of parameters that can be declared:

1. **IN** - The parameter can be referenced by the procedure or function. The value of the parameter can not be overwritten by the procedure or function.
2. **OUT** - The parameter can not be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **IN OUT** - The parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

The optional clause or replace re-creates the function. A function can be deleted using the command drop function <function name>.

## **AGGREGATE FUNCTIONS (BUILT-IN FUNCTIONS)**

Aggregate functions are statistical functions such as count, min, max etc. They are used to compute a single value from a set of attribute values of a column.

### **AVG Function**

SQL uses the AVG function to calculate the average of a column.

The syntax for using this function is,

```
Select AVG <column name> from
<table>
```

### **SUM Function**

The SUM function is used to calculate the total for a column.

The syntax is,

```
Select SUM<column name> from  
<table>
```

### **MIN Function**

SQL uses the MIN function to find the maximum value in a column.

The syntax for using the MIN function is,

```
Select MIN<column name> from  
<table>
```

### **MAX Function**

SQL uses the MAX function to find the maximum value in a column.

The syntax for using the MAX function is,

```
Select MAX <column name> from  
<table>
```

### **COUNT Function**

Another arithmetic function is COUNT. This allows us to COUNT up the number of row in a certain table.

The syntax is,

```
Select COUNT <column name> from  
<table>
```

### **VARIANCE Function**

SQL uses the VARIANCE function to find the variance value of a column.

The syntax for using the VARIANCE function is,

```
Select VARIANCE <column name> from  
<table>
```

### **STANDARD DEVIATION Function**

SQL uses the STDDEV function to find the standard deviation of a column.

The syntax for using the STDDEV function is,

```
Select STDDEV <column name> from  
<table>
```

### **GROUP BY Function**

Often applications require grouping rows that have certain properties and then applying an aggregate function on one column for each group separately. For this, SQL provides the clause *group by <group*

*column(s)>*. This clause appears after the where clause and must refer to columns of tables listed in the from clause.

The syntax for using the GROUP BY function is,

```
select <column(s)> from  
<table(s)> where <condition>  
group by <group column(s)>;
```

### **HAVING CLAUSE**

The **HAVING** clause, which is reserved for aggregate functions. The **HAVING** clause is typically placed near the end of the SQL statement, and a SQL statement with the **HAVING** clause may or may not include the **GROUP BY** clause.

```
select <column(s)> from  
<table(s)> where <condition>  
group by <group column(s)>  
[having <group condition(s)>;]
```

Note:

A query containing a group by clause is processed in the following way:

1. Select all rows that satisfy the condition specified in the where clause.
2. From these rows form groups according to the group by clause.
3. Discard all groups that do not satisfy the condition in the having clause.
4. Apply aggregate functions to each group.
5. Retrieve values for the columns and aggregations listed in the select clause.

### **STUDY- TRIGGERS**

Complex integrity constraints that refer to several tables and attributes (as they are known as assertions in the SQL standard) cannot be specified within table definitions.

Triggers, in contrast, provide a procedural technique to specify and maintain integrity constraints. Triggers even allow users to specify more complex integrity constraints since a trigger essentially is a PL/SQL procedure. Such a procedure is associated with a table and is automatically called by the database system whenever a certain modification (event) occurs on that table. Modifications on a table may include insert, update, and delete operations.

#### Structure of Triggers

A trigger definition consists of the following (optional) components:

- trigger name

create [or replace] trigger <trigger name>
--

- trigger time point

before   after
----------------

- triggering event(s)

insert or update [of <column(s)>] or delete on <table>
--

- trigger type (optional)

for each row
--------------

- trigger restriction (only for for each row triggers !)

when (<condition>)
--------------------

- trigger body

<PL/SQL block>
----------------

Below is the syntax for creating a trigger in Oracle

```
CREATE [OR REPLACE] TRIGGER <trigger_name>
{BEFORE|AFTER} {INSERT|DELETE|UPDATE} ON <table_name>
[REFERENCING [NEW AS <new_row_name>] [OLD AS <old_row_name>]]
[FOR EACH ROW [WHEN (<trigger_condition>)]]
<trigger_body>
```

Some important points to note:

- You can create only BEFORE and AFTER triggers for tables. (INSTEAD OF triggers are only available for views; typically they are used to implement view updates.)
- You may specify up to three triggering events using the keyword OR. Furthermore, UPDATE can be optionally followed by the keyword OF and a list of attribute(s) in <table\_name>. If present, the OF clause defines the event to be only an update of the attribute(s) listed after OF.

Here are some examples:

... INSERT ON R ...  
... INSERT OR DELETE OR UPDATE ON R ...  
...UPDATE OF A, B OR INSERT ON R ...

- If FOR EACH ROW option is specified, the trigger is row-level; otherwise, the trigger is statement-level.
- Only for row-level triggers:
  - The special variables NEW and OLD are available to refer to new and old tuples respectively.

**Note:** In the trigger body, NEW and OLD must be preceded by a colon (":"), but in the WHEN clause, they do not have a preceding colon! See example below.

- The REFERENCING clause can be used to assign aliases to the variables NEW and OLD.
- A trigger restriction can be specified in the WHEN clause, enclosed by parentheses. The trigger restriction is a SQL condition that must be satisfied in order for Oracle to fire the trigger. This condition cannot contain subqueries. Without the WHEN clause, the trigger is fired for each row.
- <trigger\_body> is a PL/SQL block, rather than sequence of SQL statements. Oracle has placed certain restrictions on what you can do in <trigger\_body>, in order to avoid situations where one trigger performs an action that triggers a second trigger, which then triggers a third, and so on, which could potentially create an infinite loop.

The restrictions on <trigger\_body> include:

- You cannot modify the same relation whose modification is the event triggering the trigger.
- You cannot modify a relation connected to the triggering relation by another constraint such as a foreign-key constraint.

The clause replace re-creates a previous trigger definition having the same <trigger name>.

The clause replace re-creates a previous trigger definition having the same <trigger name>. The name of a trigger can be chosen arbitrarily, but it is a good programming style to use a trigger name that reflects the table and the event(s), e.g., upd ins EMP.

A trigger can be invoked before or after the triggering event. The triggering event specifies before (after) which operations on the table <table> the trigger is executed. A single event is an insert, an update, or a delete; events can be combined using the logical connective or.



If for an update trigger no columns are specified, the trigger is executed after (before) <table> is updated. If the trigger should only be executed when certain columns are updated, these columns must be specified after the event update.

## **BASIC IDEA About Lab Program's**

**EX NO: 1      DISTRIBUTED DATABASE FOR BOOKSTORE**

**Date:**

**AIM:**

To develop and write SQL queries for a distributed database of BookStore at four sites S1, S2, S3 and S4. The Bookstores are divided into four sites by their ZIP codes.

**SQL Query:**

- a) Create tables and insert values into each table.
- b) To find the total number of books in stock where price is between \$15 and \$55

```
SQL> SELECT ISBN, Total_stock  
FROM Books  
WHERE $price > 15 and $price < 55;
```

- c) To update the book price of book No=1234 from \$45 to \$55 at site S3
- d) To find total number of book at site S2

**Eg. For Site S1** SQL> S1 = SELECT \* FROM Bookstore  
WHERE Zip<=25000;

---

**EX NO: 2**

**DEADLOCK DETECTION FOR DISTRIBUTED DATABASE**

**Date:**

**AIM:**

To implement Deadlock Detection Algorithm for Distributed Database using Wait-for Graph to check for Deadlock.

**SQL Query:**

**EX NO: 3**

**OBJECT ORIENTED UNIVERSITY DATABASE**

**Date:**

**AIM:**

To design an Enhanced Entity Relationship model for University database and to write Object Query Language (OQL) to manage the database.

**SQL Query:**

a) Draw the EER model for the university database taking into account all the constraints

i) To insert details into each object

```
create table <table> (  
  <column 1> <data type> [not null] [unique] [<column constraint>],  
  .....  
  <column n> <data type> [not null] [unique] [<column constraint>],  
  [<table constraint(s)>]  
);  
insert into <table> [(<column i, . . . , column j>)] values  
  (<value i, . . . , value j>);
```

ii) To display Employee details

iii) To display Student details

iv) To modify Person details

v) To delete Person details

b) Extend the EER model for the university database by considering the additional generalization and specialization constraints given in the question.

**EX NO: 4                      PARALLEL DATABASE – UNIVERSITY COUNSELLING FOR  
ENGINEERING COLLEGES**

**Date:**

**AIM:**

To implement University Counselling for Engineering Colleges using Parallel Database.

**PARALLEL DATABASE:**

A variety of hardware architectures allow multiple computers to share access to data, software, or peripheral devices. A Parallel Database is designed to take advantage of such architectures by running multiple instances which "share" a single physical database. In appropriate applications, a parallel server can allow access to a single database by users on multiple machines, with increased performance.

**SQL Query:**

In this exercise, the College details, Department details and Vacancy Details are maintained in 3 different sites. By using the information from these 3 sites, seats can be allocated to a student using Parallel Query Processing.

**Example 1:**

The following simple statement parallelizes the creation of a table and stores the result in a compressed format, using table compression:

```
CREATE TABLE hr.admin_emp_dept
```

```
PARALLEL COMPRESS
```

```
AS SELECT * FROM hr.employees
```

```
WHERE department_id = 10;
```

In this case, the PARALLEL clause tells the database to select an optimum number of parallel execution servers when creating the table.

### **Example 2:**

An example of parallel query with intra- and inter-operation parallelism, consider a more complex query:

```
SELECT /*+ PARALLEL(employees 4) PARALLEL(departments 4) USE_HASH(employees)
```

```
ORDERED */
```

```
MAX(salary), AVG(salary)
```

```
FROM employees, departments
```

```
WHERE employees.department_id = departments.department_id
```

```
GROUP BY employees.department_id;
```