Practical List of DBMS

1. Study of DBMS, RDBMS and ORDBMS.
2. To study Data Definition language Statements.
3. To study Data Manipulation Statements.
4. Study of SELECT command with different clauses.
5. Study of SINGLE ROW functions (character, numeric, Data functions).
6. Study of GROUP functions (avg, count, max, min, Sum).
7. Study of various type of SET OPERATORS (Union, Intersect, Minus).
8. Study of various type of Integrity Constraints.
9. Study of Various type of JOINS.
10. To study Views and Indices.
Experiment no.: 1

Aim:- Study of DBMS, RDBMS and ORDBMS.

Theory :-

Definition of DBMS
Feature s of DBMS
Advantages

Definition of RDBMS
Feature s of RDBMS
Advantages

Definition of DBMS
Feature s of DBMS
Advantages

Conclusion:
Aim:- To study Data Definition Language Statements.

Introduction:

The schema data Definition Language (DDL) is used for describing a database which maybe shared by many programs written in many languages. This description is it terms of the names and characteristics of the data items, data aggregates, records, areas. And sets included in the database, and the relationships that exist and must be maintained between occurrences of these elements in the database. I data items. A data item is an occurrence of the smallest unit of named data. It is represented in a database by a value.

Theory:-

SQL schema update statements (DDL)

Delete or Modify table

ALTER TABLE statement will allow you to add new column or constraint, modify existing column or constraint, and drop existing constraint after the table is created. The following will add a middle name column to the employee’s table.

```
ALTER TABLE employees
    ADD middle_name VARCHAR2 (25);
```

The statement bellow modifies the last name to required column. This statement works when there no null last name exist now.

```
ALTER TABLE employees
    ADD last_name VARCHAR2 (25);
```

This statement deletes the above column.

```
ALTER TABLE employees
    ADD middle_name VARCHAR2 (25);
```

Adding and deleting constraints are similar to these of the column. Here is an statements to add a forgin key to employee table.
To delete a column or constraints use DROP followed by the column or constraints name in the position of ADD or MODIFY. Constraints can be turned off temporarily instead of dropping it, by using DISALBLE/ENABLE clause within the alter table statements. The bellow statements turn off the primary key for employee table. ENABLE is used when turning it on.
ALTER TABLE employees DISALBLE PRIMARY KEY;

Create table statement

Create table-statement :
CREATE TABLE table-name (column-definition comma – list)

Drop table statement

don- table – statement :
DROP TABLE table-name

Alter table statement

alter- table-statement:
ALTER TABLE table-name {ADD add-column-set [DROP drop-column-set] [DROP drop-column-set]}

Conclusion :
Experiment no.: 3

Aim:- To study Data Manipulation Languages statements.

Theory:

The basic data Manipulation functions in these specifications include the functions required to: Select records Present recodes to the run unit. Add new records and relationships. Change existing records and relationships. Remove existing records and relationships.

SQL Data update statements (DML)

---

Insert statement

insert -statement:

INSERT INTO table – name [column- identifier-comma-list)] VALUES (column-value-comma-list)

---

Delete statement

delete - statement :

DELETE FROM table-name [WHERE search- condition]

---

Update statement

update - statement:

UPDATE table-name SET update- column-list [WHERE search- condition]
*Adding new rows to database table*

We created the table, define the constraints and now they are ready to be populated was data INSERT command is used to populate the table with records. Here is a statement that adds a row or record to product table. This type of statement needs all the column to be exact order. This is a poor way to insert values into a table.

\[
\text{INSERT INTO Product} \\
\text{VALUES (‘10000000’,’Printer’,’Inkjet colour Printer’,’120’,’80’):}
\]

Here is a more favorable and controllable way to insert values into the a table.

\[
\text{INSERT INTO Product (product_id, name, description, price, cost)} \\
\text{VALUES (‘10000000’,’Printer’,’Inkjet colour Printer’,’120’,’80’):}
\]

**Conclusion:**
Aim:- Study of SELECT COMMAND with different clauses.

Theory:

SQL SELECT

What do we use SQL for? Well, we use it to select data from the tables located in a database. Immediately, we see two keywords: we need to SELECT information FROM a table. We have the most basic SQL structure:

SELECT “column name” FROM “table name”

To illustrate the above example, assume that we have the the following table:

<table>
<thead>
<tr>
<th>Store_name</th>
<th>Sales</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Los Angeles</td>
<td>$1500</td>
<td>Jan-05-1999</td>
</tr>
<tr>
<td>San Diego</td>
<td>$250</td>
<td>Jan-07-1999</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>$300</td>
<td>Jan-08-1999</td>
</tr>
<tr>
<td>Boston</td>
<td>$700</td>
<td>Jan-08-1999</td>
</tr>
</tbody>
</table>

We shall use this table as an example throughout the tutorial (This table will appear in all sections). To select all the stores in this table, we key in,

SELECT Store_name FROM Store_ Information

Result:

Store name
Los Angeles
San Diego
Los Angeles
Boston
Multiple column names can be selected, as well as multiple table names.

**SQL DISTINCT**

The SELECT keyboard allows us to grab all information from a column (or columns) on a table. This, of course, necessarily mean that there will be redundancies. What if we only want to select each DISTINCT elements? This is easy to accomplish in SQL. All we need to do is to add DISTINCT after SELECT. The syntax is as follows:

```
SELECT DISTINCT "column_name"
FROM "table_name"
```

For example, to select all distinct stores in Table Store_Information,

**Table Store_Information**

<table>
<thead>
<tr>
<th>Store_name</th>
<th>Sales</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Los Angeles</td>
<td>$1500</td>
<td>Jan-05-1999</td>
</tr>
<tr>
<td>San Diego</td>
<td>$250</td>
<td>Jan-07-1999</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>$300</td>
<td>Jan-08-1999</td>
</tr>
<tr>
<td>Boston</td>
<td>$700</td>
<td>Jan-08-1999</td>
</tr>
</tbody>
</table>

we key in,

```
SELECT DISTINCT store_name FROM Store_Information
```

**Result:**

```
Store name
Los Angeles
San Diego
Los Angeles
Boston
```

Retrieving data from a tables

The SELECT statement can be used to retrieve data from any or all columns from a table. Say that you have a table called product, which consists product name, description, price, and etc. This is the statement to retrieve all the columns from the table product.

```
SELECT * FROM product
```

SELECT – specifies retrieval command plus what to select.

* - stands for everything. In this case, In stands for every columns in a table.
FROM – specifies the table required to retrieve information from.

Now, say that we want to retrieve only the product name, description and price not everything. Here is how you would accomplish it.

```
SELECT name, description, price FROM product;
```

How about if we want search product by specific characteristics? That is when WHERE clause becomes handy. Where clause is used when only a subset of all the rows in a table is required. Here is how you would list the name the price of the products has the price valueless than $100.

```
SELECT name, price FROM product WHERE price < 100;
```

Here are operators that may be used in the WHERE clauses;
- = equal to
- >, < greater then, less than
- >=, <= greater then or equal to, less than or equal to.
- IS NULL means no value stored in this field.
- WHERE price <= 50; will display product selling c or less
- WHERE price = 50; will display product selling at $50
- WHERE description IS NULL; will display product that has no description
- WHERE description IS NOT NULL; will display product that has descriptions

There are times you want list items by conditioning two or more characteristics. For example, if you want list product that are more than $100 or less than $50, here is how you would accomplish it.

```
SELECT * FROM product WHERE price < 50 OR price > 100;
```

The above statements will display product that are less than or greater then $1000. Logical operators AND and OR is used in SQL as we them in English language. And will result true when both conditions are met while OR will result true when either conditions is true.

```
SELECT name, description, price FROM product
WHERE( name = ‘Printer’ OR name = ‘Scanner’) AND price<= 100;
```
The above query will find all the printer and scanners that are less than or equal $100
Note how single quotations are used. IN operator can be used instead of OR. Here is
an equivalent query that uses IN operator.

```
SELECT name, description, price FROM product
WHERE name IN( ‘Printer’ ,‘Scanner’) AND price<= 100;
```

When you finding an string value, you can't sure the correct spelling. For example Printer
can also be printer, PRINTER, or printers. There are more ways to solve this problem
include, changing the column to upper case characters, or using LIKE statements. The
LIKE operator provides pattern matching for character data and uses that is position –
detection. Here is an example that uses like operator.

```
SELECT* FROM product WHERE name LIKE ‘scan% ;
```

This will display all the products that start scan. It could be scan, scanner, scanners, scan
camera, etc. To list all the items that has the word scan in it, we rephrase the statement
like this; WHERE name LIKE ‘%scan%’

BETWEEN operator can be used in numeric values to compare column against a range of
inclusive values. This is one example all the products between $50 and $100 inclusive.

```
SELECT name, description, price FROM product
WHERE price BETWEEN 50 AND 100;
```

The retrieved data may not be in any specific order unless other ways specified. Data can
be sorted using ORDER BY clause in the select statement.

```
SELECT name “Product Name”, description “Product description”, price
FROM product
ORDER BY price DESC, name ASC
```

This example simply lists product name, description, and the price sorted in descending
order and name in ascending order. The column aliases display more
Understandable column headers instead of actual column names.
Retrieving Calculated fields

You can perform mathematical calculations while retrieving data from tables or views. The math operators for SQL are same as those of other languages, like +,-, *, /, etc. Say that you want calculates and display the profit of each item in the product table. The profit price minus cost. Here is how you would do that:

SELECT Product_id, name, price, cost, price - cost
FROM product;

This example displays list of each product’s information plus calculated filed that displays item profit [price – cost]

Conclusion:
Aim: Study of single row functions (character, numeric, data functions)

Theory:

**Number Functions**

You can manipulate retrieved numbers by rounding up to specific precessions or perform extra calculation functionalities. SQL has several number functions that you can use to manipulate retrieved numbers. The following are list of functions and example of each.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
</table>
| ABS(number)            | Returns the absolute value of a number           | SELECT ABS(price – cost)                     | If (price – cost) = 3.5  
|                        |                                                  |                                              | Result is 3.5         |
| CEIL(number)           | Rounds up the number to next integer             | SELECT CEIL (price – cost) FROM product WHERE product_id = 1: | If (price – cost) = 3.5  
|                        |                                                  |                                              | Result is 4           |
| FLOOR(number)          | Rounds down the number to next integer           | SELECT FLOOR (price – cost) FROM product WHERE product_id = 1: | If (price – cost) = 3.5  
|                        |                                                  |                                              | Result is 3           |
| MOD(number, divisor)   | Returns the remainder of a division              | SELECT MOD (price,2) FROM product WHERE product_id = 1: | If price = 3  
|                        |                                                  |                                              | Result is 1           |
| POWER(number, power)   | Returns result of number raised specified power  | SELECT POWER (price,2) FROM product WHERE product_id = 1: | If price = 3  
|                        |                                                  |                                              | Result is 9           |
| SIGN(number)           | Tests if the number negative or positive and returns 1 or -1 | SELECT SIGN (price – cost) FROM product WHERE product_id = 1 | If (price – cost) = 3.51  
|                        |                                                  |                                              | Result is -1          |
| ROUND(number, precision)| Returns the a number rounded to the specified precision | SELECT ROUND (price – cost) FROM product WHERE product_id = 1: | If (price – cost) = 3.5122212  
|                        |                                                  |                                              | Result is 3.51        |
| SQRT(number)           | Returns the square root of a number              | SELECT SQRT (price) FROM product WHERE product_id = 1: | If(price) = 2  
|                        |                                                  |                                              | Result is 2           |
| TRUNC(number, precision)| Returns a number truncated to the specified precision | SELECT TRUNC (price- cost,1) FROM product WHERE product_id = 1: | If (price – cost) = 3.5133  
<p>|                        |                                                  |                                              | Result is 3.5         |</p>
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCAT(string1, string2)</td>
<td>Concatenates two strings</td>
<td>SELECT CONCAT (frist_name, last_name) FROM employees;</td>
<td>If (frist_name = ‘John’, last_name = ‘Morgan’) Result is John, Morgan</td>
</tr>
<tr>
<td>INITCAP(string)</td>
<td>Returns strings with first letter in upper case</td>
<td>SELECT INITCAP (frist_name) FROM employees;</td>
<td>If (frist_name = ‘John’) Result is John</td>
</tr>
<tr>
<td>LENGTH (string)</td>
<td>Returns an integer representing the string length</td>
<td>SELECT LENGTH (frist_name) FROM employees;</td>
<td>If (frist_name = ‘John’) Result is 4</td>
</tr>
<tr>
<td>LPAD (string, #, Padding_char)</td>
<td>Returns the string padding Characters added to the left</td>
<td>SELECT LPAD (price, 7, '*' ) FROM product WHERE product_id = 1;</td>
<td>If price = 100 Result is **** 100</td>
</tr>
<tr>
<td>RPAD (string, #, Padding_char)</td>
<td>Returns the string padding Characters added to the right</td>
<td>SELECT RPAD (price, 7, '*' ) FROM product WHERE product_id = 1;</td>
<td>If price = 100 Result is 100****</td>
</tr>
<tr>
<td>LTRIM(string, serachSrting)</td>
<td>Remove character from left/ right of char</td>
<td>SELECT LTRIM (frist_name, 'j') FROM employees; WHERE employees _id = 1;</td>
<td>If (frist_name = ‘John’) Result is ‘John’</td>
</tr>
<tr>
<td>RTRIM(string, serachSrting)</td>
<td>Remove character from left/ right of char</td>
<td>SELECT LTRIM (frist_name, 'j') FROM employees; WHERE employees _id = 1;</td>
<td>If (frist_name = ‘John’) Result is ‘John’</td>
</tr>
<tr>
<td>REPLACE (string, serachSrting, replacement)</td>
<td>REPLACE returns char with every occurrence of search String replacement String</td>
<td>SELECT REPLACE (frist_name, ‘j’”H”) FROM employees; WHERE employees _id = 1;</td>
<td>If (frist_name = ‘John’) Result is ‘Hohn’</td>
</tr>
<tr>
<td>SUBSTR(string, start, length)</td>
<td>Returns substring starting at start and of specified length</td>
<td>SELECT SUBSTR (frist_name, ‘1’”3”) FROM employees; WHERE employees _id = 1;</td>
<td>If (frist_name = ‘John’) Result is ‘Joh’</td>
</tr>
<tr>
<td>UPPER(string)</td>
<td>Returns string with all upper/ lower case characters</td>
<td>SELECT UPPER (frist_name,) FROM employees; WHERE employees _id = 1;</td>
<td>If (frist_name = ‘John’) Result is ‘JOHN’</td>
</tr>
</tbody>
</table>
Concatenation and column alias

You can manipulate the retrieved data by joining two columns or changing the column heading. The default heading is actual column name. For example, when you select product_id from product, the heading is product id. We can change the column heading to “product ID” by using following syntax:

```
SELECT column_name " column_alias"
FROM table_name;
```

The following example change and display column heading of product_id as Product ID, and column heading of name as Product Name.

```
SELECT product_id "Product Name FROM product:
```

This example changes the column heading of product_id and name to proper more understandable column headings

We can also concatenate two or more fields together in select statement. For instance, if we want join first_name and last_name together to make up full_name, we can easily do this using following syntax:

```
SELECT column_name || column_name FROM table_name; or SELECT
CONCAT(column_name, column_name) FROM table_name; Both methods accomplish
same results. The following example retrieves concatenated first_name and last name
with space between and column heading of Full Name:
```

```
SELECT frist_name ||' ||last_name "Full Name" FROM employees;
```

Single-Row Character Manipulation Functions

There are number of string manipulation functions available for use of character output like the CONCAT we saw. The following are list of those functions with description, example and result:

Single-Row Date Functions

All sql data can manipulated on retrieval or displayed as you desire it to be displayed. The date functions we are discussing next can be used to manipulate date data types. For example, if you want to add few days, month, years to the current date or format the output of the dates retrieved by select statement. This can done using date functions. The following are some list of date functions available in sql:
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSDATE</td>
<td>Returns current date</td>
<td>SELECT sysdate FROM dual;</td>
<td>If (today = 1.01 2002) Result is 1-jan-02</td>
</tr>
<tr>
<td>ADD_MONTHS(Date, months_to_add)</td>
<td>Returns date retrieved date added to specified month</td>
<td>SELECT ADD_MONTHS (sysdate,3) FROM dual;</td>
<td>If (SYSDATE = 4/30 200) Result is 4/30 '2002'</td>
</tr>
<tr>
<td>LAST_DAY (Date)</td>
<td>Returns date that is last day of the month</td>
<td>SELECT LAST_DAY (sysdate) FROM dual;</td>
<td>If (SYSDATE = 1/20 200) Result is 1/31 '2002'</td>
</tr>
<tr>
<td>MONTHS_BETWEEN(Date1, Date2)</td>
<td>Returns months between two dates specified months</td>
<td>SELECT MONTHS_BETWEEN(TO_DATE('02-02-2002'), SYSDATE) FROM dual;</td>
<td>If (SYSDATE = 1, 2002 Result is 1.0322.</td>
</tr>
</tbody>
</table>

**Conclusion:**
Experiment no.: 6

Aim :- Study of GROUP FUNCTIONS

Theory:-

Group Functions

An SQL group function or aggregate functions performs an operation on a group of rows and returns a single result. You may want retrieve group of item-prices and return total- price. This type of scenario is where you would use a group functions. The following table is summary of some SQL group function & query examples.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Query Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG (fieldname)</td>
<td>Returns average value of a column</td>
<td>SELECT avg(price) FROM inventory;</td>
</tr>
<tr>
<td>COUNT (fieldname)</td>
<td>Returns number of items in Table or queried items</td>
<td>SELECT count(product_id) FROM product;</td>
</tr>
<tr>
<td>COUNT (*)</td>
<td>Returns minimum value of Column</td>
<td>SELECT min(price) FROM inventory;</td>
</tr>
<tr>
<td>MIN(fieldname)</td>
<td>Returns maximum value of Column</td>
<td>SELECT max(price) FROM inventory;</td>
</tr>
<tr>
<td>SUM(fieldname)</td>
<td>Returns total value of Column</td>
<td>SELECT sum(price) FROM inventory;</td>
</tr>
</tbody>
</table>

To use a group function in a SQL query, list the function name followed by numeric column name within parentheses. AVG averages the column, COUNT counts the number of items, MAX returns maximum number of the column, and MIN returns minimum number of the column.

The following is query to retrieve total price, average price, maximum price, and minimum price from the table “product” assuming the product table has the following values.

<table>
<thead>
<tr>
<th>Product ID</th>
<th>Name</th>
<th>Description</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000000</td>
<td>Printer</td>
<td>Inkjet 300 colour Printer</td>
<td>120</td>
</tr>
<tr>
<td>1000000001</td>
<td>Printer</td>
<td>1220XI Inkjet Printer</td>
<td>200</td>
</tr>
<tr>
<td>1000000002</td>
<td>Printer</td>
<td>Photo 890 Inkjet Printer</td>
<td>250</td>
</tr>
<tr>
<td>1000000003</td>
<td>Printer</td>
<td>Photo 890 Inkjet Printer</td>
<td>300</td>
</tr>
</tbody>
</table>
SQL statements.

SELECT sum(price)
FROM product;

This statement will returns the total amount for the column price which is 870.

SELECT avg(price)
FROM product;

This statement will returns the average amount for the column price which is 870/4 or 217.50.

SELECT max(price)
FROM product;

This statement will returns the maximum amount for the column price which is 300.

SELECT min(price)
FROM product;

This statement will returns the minimum amount for the column price which is 120.

SELECT count (*)
FROM product;

This statement will returns the number of items in table which is 4.

**GROUP BY Clause with Group Functions**

**Group By** is used to categorize the retrieved data. For example, you may wight want list sales of each product identified by product id. To do this, the following is Group By.

example that lists.

**Conclusion:**
UNION ALL Example

The UNION operator returns only distinct rows that appear in either result, while the UNION ALL operator returns all rows. The UNION ALL operator does not eliminate duplicate.

```sql
SELECT product_id FROM order_items
UNION
SELECT product_id FROM inventories;
```

```sql
SELECT location_id FROM locations
UNION ALL
SELECT location_id FROM departments;
```

A `location_id` value that appears multiple times in either or both queries (such as ‘1700’) is returned only once by the UNION operator, but multiple times by the UNION ALL operator.

INTERSECT Example

The following statement combines the result with the INTERSECT operator, which returns only those rows returned by both queries:

```sql
SELECT product_id FROM inventories
INTERSECT
SELECT product_id FROM order_items;
```

MINUS Example

The following statement combines results with the MINUS operator, which returns only rows returned by the first query but not by the second:

```sql
SELECT product_id FROM inventories
MINUS
SELECT product_id FROM order_items;
```

Conclusion:
AIM: Study of various types of SET OPERATORS

Theory:

The UNION [ ALL], INTERSECT, MINUS Operators

You can combine multiple queries using the set operators UNION, UNION ALL, INTERSECT and MINUS. ALL set operators have equal precedence. If a SQL statement contains multiple set operators, than Oracle evaluates them, from the left to right if no parentheses explicitly specify another order.

The corresponding expressions in the select lists of the component queries of a compound query must match in number and datatype. If component queries select character data, then the datatype of the return values are determined as follows:
- If both queries select values of datatype CHAR, then the returned values have datatype CHAR
- If either or both of the queries select values of datatype VARCHAR2, than the returned values have datatype VARCHAR2.

UNION Example

The following statement combines the result with the UNION operator, which eliminates duplicate selected rows. This statement shows that you must match datatype (using the TO_CHAR function) when columns do not exist in one or the other table:

```sql
SELECT location_id, department name "Department",
TO_CHAR (NULL) "Warehouse" FROM departments
UNION
SELECT location_id, TO_CHAR (NULL) "Department", warehouse_name
FROM warehouse;
```

<table>
<thead>
<tr>
<th>LOCATION_ID</th>
<th>Department</th>
<th>Warehouse</th>
</tr>
</thead>
<tbody>
<tr>
<td>1400</td>
<td>IT</td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td></td>
<td>Southlake, Texas</td>
</tr>
<tr>
<td>1500</td>
<td>Shipping</td>
<td></td>
</tr>
<tr>
<td>1500</td>
<td></td>
<td>San Francisco</td>
</tr>
<tr>
<td>1600</td>
<td></td>
<td>New Jersey</td>
</tr>
<tr>
<td>1700</td>
<td>Accounting</td>
<td></td>
</tr>
<tr>
<td>1700</td>
<td>Administration</td>
<td></td>
</tr>
<tr>
<td>1700</td>
<td>Benefits</td>
<td></td>
</tr>
<tr>
<td>1700</td>
<td>Construction</td>
<td></td>
</tr>
</tbody>
</table>
Aim:- Study of various types of Integrity Constraints.

Theory:
CREATE TABLE branch
(name varchar (10) PRIMARY KEY,
city varchar (20),
director varchar (20),
assets number)

**Integrity Constraints**
attribute Constraint
CREATE TABLE t1
(
c integer PRIMARY KEY,
d integer NOT NULL,
e integer)

**Integrity Constraints**
attribute Constraint
CREATE TABLE t1
(
c integer PRIMARY KEY,
d integer NOT NULL UNIQUE,
e integer)

**SQL DDL**, creation example
CREATE TABLE branch
(name varchar (10) PRIMARY KEY,
city varchar (20),
director
archar (20) NOT NULL UNIQUE,
assetsnumber)

**Integrity Constraints**
attribute Constraint
CREATE TABLE t1
(
c integer PRIMARY KEY,
d integer NOT NULL UNIQUE,
e integer REFERENCES t0(f)
f must be a primary key of t0
**SQL DDL**, creation example

```
CREATE TABLE branch
  ( branch_name varchar(10),
    city
    varchar (20),
    employee
    varchar (20),
    REFERENCES staff(name))
```

**Integrity Constraints**

attribute Constraint

```
CREATE TABLE t1
  (c integer PRIMARY KEY,
  d integer NOT NULL UNIQUE,
  e integer REFERENCES t0(f)
  CHECK (e>18))
```

**Conclusion:**
Aim: Study of various types of JOINs

Theory:

Joints

Using SQL Joints, you can retrieved data more than one table or views using the keys [Primary & foreign] references.

The syntax for joining multiple tables is as follows:

```
SELECT column,
FROM table1, table2
WHERE table1, join_column= table2, join_column;
```

The SELECT statement contains the columns to retrieve and may come from two or more tables. If the selected column exists both tables, specify the table [table.column]. You have to specify the tables in the FROM clause and the joint condition is done in the WHERE clause. In the WHERE clause, you need the table name and dot followed by column name. The column name in the WHERE clause is the joint column [keys]. AND & OR also normally used to make multiple joint conditions. There are about four basic types of joints, Equality Joints, Outer Joints, Staff Joints, & Inequality Joints.

Equality Joints

Equality Joints happens when two table are joined based on values in one table being equal to values in anther table. For example, product table and inventory table may have joint column of product_id as it show in the following tables.

<table>
<thead>
<tr>
<th>Product ID</th>
<th>Name</th>
<th>Description</th>
<th>Price</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000000</td>
<td>Printer</td>
<td>Inkjet 300 colour Printer</td>
<td>120</td>
<td>80</td>
</tr>
<tr>
<td>1000000001</td>
<td>Printer</td>
<td>1220XI Inkjet Printer</td>
<td>200</td>
<td>130</td>
</tr>
<tr>
<td>1000000002</td>
<td>Printer</td>
<td>Photo 890 Inkjet Printer</td>
<td>250</td>
<td>200</td>
</tr>
<tr>
<td>1000000003</td>
<td>Printer</td>
<td>Photo 890 Inkjet Printer</td>
<td>300</td>
<td>270</td>
</tr>
</tbody>
</table>
Inventory Table

<table>
<thead>
<tr>
<th>Product ID</th>
<th>qty_on_hand</th>
<th>qty_on_order</th>
<th>min_req</th>
<th>mx_req</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000000</td>
<td>20</td>
<td>0</td>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>100000001</td>
<td>10</td>
<td>5</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>100000002</td>
<td>2</td>
<td>10</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>100000003</td>
<td>1</td>
<td>15</td>
<td>1</td>
<td>15</td>
</tr>
</tbody>
</table>

The query to perform equality joints might look like this:

```
SELECT product. product_id, name, price, qty_on_hand, qty_on_order
FROM product, inventory
WHERE product. product_id = inventory. product_id
```

Product table is specified in the SELECT statements to issue the product the product_id and the reason is, product_id exists both tables and if you don’t specified which table to select from, you will receive ambiguous error. The query will select all the selected rows from both tables since there is always product_id equal to product_id in the other table.

**Outer Joints**

**Outer Joints** is joint condition where all the rows of a table are selected along with then matching rows in the other table. For example you might want select all your customers along with their orders if they have orders. + sign is used in the WHERE clause beside the child table within parentheses. The following is SQL joint statements to select every customer in the customer table along with their orders if they have orders and if they.

```
SELECT customer. customer_id, firstname, lastname, item_id, qty_ordered, price
FROM customer, order
WHERE customer. customer_id = order. customer_id (+);
```

**Self Joints**

**Self Joints** is joint of table by it self. For example, if you want retrieve customers whom ordered same products twice or more assuming there is num_order column that keeps track the number of orders customers made. Here is how you would do this using a self joints:

```
SELECT 01. customer_id
FROM order 01, order 02
WHERE 01. item_id = 02. item_id AND 01.num_order>1;
```
This query simply created two table alias, 01 and 02 which represents two copies of the table order then compares if item_id exists both table when order is placed two or more times by a customer.

**Inequality Joints**

**Inequality Joints** is when each record in one table is joined with every record in the second table using operators $<$, $>$, $=,$ etc. to define an inequality condition. It’s opposite of inner joints. This type of joint is rarely used since joint. This type of joint is rarely used since joint columns are keys & inequality comparison of the keys has no meaningful applications.

```sql
SELECT first_name || '' || last_name " Full Name"
FROM customer, order
WHERE customer.customer_id < order.customer_id;
```

**Conclusion:**
Experiment no: 10

Aim: To Study views and indices.

Indexes:

Indexes in the store database are B+tree structures with a fixed size key. This means that text keys may easily generate large indexes, and maximum length text column cannot be indexed.

Keys in DBMS may be truncated by specifying the length of a text column to be used it the index key. Such a key column must always be the last one in the key, to ensure that the index provides a true ordering. If a text key column does not supply a truncation length, the full length of the table column is used.

Apart from Binary and LongBinary columns, all types can be used in an index key including LongText8 and LongText 16, although these must supply a truncation length (as they have no maximum length) and must therefore be the last key columns in the key.

Create index statements

Create-index- statement : CREARE [UNIQUE] INDEX index- name ON table-name

VIEWS

SQL View is a virtual or temporary table. Sql view is similar to the query in access so that you retrieve data from a table and store it in a view. Views are often created because of security reasons or to retrieve and store data from different tables. You can create a view same way as a table. Select statements is used to retrieve data from a table. Here is an example of creating a view:

CREATE VIEW product_view AS
SELECT name, description
FROM product;

The name of the view we just created is product_view, which selects two columns. Name and desc from product table. The values from these two columns will be stored in this view.

Conclusion :-
UNIT VI
Database System Architectures and Distributed Database: Centralized Systems, Client/Server Systems, Parallel Systems, Distributed data storage, Network transparency Distributed query processing, Distributed transaction model, Commit Protocols, Concurrency controls, Deadlock handling, Multidatabase Systems.

Text Books :
1. Database System Concepts: Korth, Silbersehatz: Megraw- Hill
2. Database Management System : Majumdar & Bhattacharya
3. Principles of Database : Jeffrey D. Ullman
4. An Introduction To Database Systems : C. J. Data (Adison Wesley - Nerosa)