**AMRITA VISHWA VIDYAPEETHAM CHENNAI CAMPUS**

**SECOND Year. B.E (Computer Science and engineering)**

**DATABASE MANAGEMENT SYSTEMS**

**Work Book**

**(From Academic year 2025**)

**Name: S DEEPAK BHARATHWAJ**

**College Name: Amrita vishwa vidyapeetham, CHennai**

**Roll No.: ch.sc.u4cse24015 DEPARTMENT: CSE - core**

**Academic Year: 2025-2026**

Preface to the First Edition

The new syllabus For Second Year B.Tech Computer Science is implemented from the academic year 2023. It is absolutely necessary and essential that all the Computer Science practicals be conducted on Open-Source Operating System like Linux or Windows. All the practical’s related To DBMS needs to be conducted using MySQL an open-source software.

.

It is mandatory to carry the completed and duly Signed lab activity sheets for the practical examinations.

Editors:

Dr a padmavathi

Introduction

1. About the work book

This workbook is intended to be used by second year B.tech. computer science students for the Database Management Systems laboratory courses in their curriculum. In Computer Science, hands-on laboratory experience is critical to the understanding of theoretical concepts studied in the theory courses. This Workbook provides the requisite background material as well as numerous computing problems covering all difficulty levels.

The objectives of this book are

* Defining clearly the scope of the course
* Bringing uniformity in the way the course is conducted across different campuses.
* Continuous assessment of the course
* Bring in variation and variety in the experiments carried out by different students in a batch
* Providing ready reference for students while working in the lab
* Catering to the need of slow learners and fast learners

2. How to use this work book?

This workbook is mandatory for the completion of the laboratory course. It is a measure of the performance of the student in the laboratory for the entire duration of the course.

2.1 Instructions to the students

Please read the following instructions carefully and follow them

1) Carry this book every time you come to the lab for DBMS practical.

2) Prepare yourself beforehand for the exercise by reading the material mentioned under.

3) If the self-activity exercise or assessment work contains any blanks such as --------, get them filled by your instructor.

4) Instructor will specify which problems you are to solve by ticking bo

5) you will be assessed for each exercise on a scale of 5

i) not done 0 ii) incomplete 1

iii) late complete 2 iv) needs improvement 3

v) complete 4 vi) well done 5

2.2. **Instruction to the Instructors**

1. Explain the related concepts by demonstrating the software and worksheet.
2. Fill in the blanks with different values for each student
3. Choose appropriate problems to be solved by student by ticking box
4. Make sure that students follow the instruction as given above
5. After a student completes a specific set, the instructor has to verify the outputs and sign in the provided space after the activity.
6. Assignment carried out by a student must be evaluated on a scale of 5 as specified above by ticking appropriate box.
7. The value should also be entered on assignment completion page of the respective lab course

**2.3. Instructions to the lab administrator**

Must ensure appropriate hardware and software is installed in each and every computer and ensure that system is allotted to the students properly.

The operating system and software requirements on server side and also client side are as given below

Server-side Operating System - \* Ubuntu Linux \*

Servers side software to be installed) – postgresql / mysql

Client-side Operating System - \* Ubuntu Linux \*

Client-side software to be installed - postgresql/mysql

**Activity Completion Sheet**

|  |  |  |
| --- | --- | --- |
| **database management systems** | | |
| **sno** | **lIST OF experiments** | **MARKS** |
| 1 | Ddl cOMMANDS – CREATION OF TABLE AND SIMPLE QUERIES |  |
| 2 | DML COMMANDS – mANIPULATING THE DATABASES |  |
| 3 | sql cONSTRAINTS |  |
| 4 | Implementation of Arithmetic / logical Operations, Sorting and Grouping. |  |
| 5 | Builtin operations |  |
| 6 | USING THE SET OPERATIONS |  |
| 7 | aGGREGATE FUNCTIONS |  |
| 8 | sql jOINS |  |
| 9 | SUBQUERY |  |
| 10 | PL / SQL |  |
| 11 | Creating VIEWS, SYNONYMS, INDEX, SEQUENCE |  |
| 12 | advaned databases (mongo db) |  |

This is to certify that, Mr./Miss. Roll No. of I year B.Tech. (Computer Science and Engineering) has successfully completed out of practical’s satisfactorily during the academic year 2024 - 2025.

**Practical In-charge Chairperson**

**Database Management Systems Lab**

**Definition of a Relational Database**

A relational database is a collection of relations or two-dimensional tables.

**Terminologies Used in a Relational Database**

1. A single ROW or table representing all data required for a particular employee. Each row should be identified by a primary key which allows no duplicate rows.
2. A COLUMN or attribute containing the employee number which identifies a unique employee. Here Employee number is designated as a primary key ,must contain a value and must be unique.
3. A column may contain foreign key. Here Dept\_ID is a foreign key in employee table and it is a primary key in Department table.
4. A Field can be found at the intersection of a row and column. There can be only one value in it. Also it may have no value. This is called a null value.

|  |  |  |  |
| --- | --- | --- | --- |
| EMP ID | FIRST NAME | LAST NAME | EMAIL |
| 100 | King | Steven | Sking |
| 101 | John | Smith | Jsmith |
| 102 | Neena | Bai | Neenba |
| 103 | Eex | De Haan | Ldehaan |

**Relational Database Properties**

**A relational database :**

* Can be accessed and modified by executing structured query language (SQL) statements.
* Contains a collection of tables with no physical pointers.
* Uses a set of operators

**Relational Database Management Systems**

RDBMS refers to a relational database plus supporting software for managing users and processing [SQL](http://cplus.about.com/od/glossar1/g/sqldefinition.htm) queries, performing backups/restores and associated tasks.

(Relational Database Management System) Software for storing data using SQL (structured query language). A relational database uses SQL to store data in a series of tables that not only record existing relationships between data items, but which also permit the data to be joined in new relationships. SQL (pronounced 'sequel') is based on a system of algebra developed by E. F. Codd, an IBM scientist who first defined the relational model in 1970. Relational databases are optimized for storing transactional data, and the majority of modern business software applications therefore use an RDBMS as their data store. The leading RDBMS vendors are Oracle, IBM and Microsoft.

**SQL Statements**

1. Data Retrieval(DR)
2. Data Manipulation Language(DML)
3. Data Definition Language(DDL)
4. Data Control Language(DCL)
5. Transaction Control Language(TCL)

|  |  |  |
| --- | --- | --- |
| **TYPE** | **STATEMENT** | **DESCRIPTION** |
| DR | SELECT | Retrieves the data from the database |
| DML | 1.INSERT  2.UPDATE  3.DELETE  4.MERGE | Enter new rows, changes existing rows, removes unwanted rows from tables in the database respectively. |
| DDL | 1.CREATE  2.ALTER  3.DROP  4.RENAME  5.TRUNCATE | Sets up, changes and removes data structures from tables. |
| TCL | 1.COMMIT  2.ROLLBACK  3.SAVEPOINT | Manages the changes made by DML statements. Changes to the data can be grouped together into logical transactions. |
| DCL | 1.GRANT  2.REVOKE | Gives or removes access rights to both the oracle database and the structures within it. |

**Data Types**

* 1. **Character Data types:** 
     + Char – fixed length character string that can varies between 1-2000 bytes
     + Varchar / Varchar2 – variable length character string, size ranges from 1-4000 bytes.it saves the disk space(only length of the entered value will be assigned as the size of column)
     + Long - variable length character string, maximum size is 2 GB
  2. **Number Data types :** Can store +ve,-ve,zero,fixed point, floating point with 38 precission.
     + Number – {p=38,s=0}
     + Number(p) - fixed point
     + Number(p,s) –floating point (p=1 to 38,s= -84 to 127)
  3. **Date Time Data type:** used to store date and time in the table.
     + DB uses its own format of storing in fixed length of 7 bytes for century, date, month, year, hour, minutes, and seconds.
     + Default data type is “dd-mon-yy”
     + New Date time data types have been introduced. They are

TIMESTAMP-Date with fractional seconds

INTERVAL YEAR TO MONTH-stored as an interval of years and months

INTERVAL DAY TO SECOND-stored as o interval of days to hour’s minutes and seconds

* 1. **Raw Data type:** used to store byte oriented data like binary data and byte string.
  2. **Other :** 
     + CLOB – stores character object with single byte character.
     + BLOB – stores large binary objects such as graphics, video, sounds.
     + BFILE – stores file pointers to the LOB’s.

**ACTIVITY SHEET -01 DATE:**

**DDL COMMANDS – CREATING AND MANAGING TABLES**

**Objective**

After the completion of this exercise, students should be able to do the following:

* Create tables
* Describing the data types that can be used when specifying column definition
* Alter table definitions
* Drop, rename, and truncate tables

**Reading:**

You should read this before starting the exercise.

**Naming Rules**

Table names and column names:

* + Must begin with a letter
  + Must be 1-30 characters long
  + Must contain only A-Z, a-z, 0-9, \_, $, and #
  + Must not duplicate the name of another object owned by the same user
  + Must not be an oracle server reserve words
  + 2 different tables should not have same name.
  + Should specify a unique column name.
  + Should specify proper data type along with width
  + Can include “not null” condition when needed. By default it is ‘null’.

**The CREATE TABLE Statement**

**Table:** Basic unit of storage; composed of rows and columns

**Syntax: 1** Create table table\_name (column\_name1 data\_ type (size) , column\_name2 data\_ type (size)….);

**Syntax: 2** Create table table\_name (column\_name1 data\_ type (size) constraints, column\_name2 data\_ type constraints …);

**Example:**

Create table employlees ( employee\_id number(6), first\_name varchar2(20), ..job\_id varchar2(10), CONSTRAINT emp\_emp\_id\_pk PRIMARY KEY (employlee\_id));

**Creating a table by using a Sub query**

**SYNTAX**

CREATE TABLE table\_name(column\_name type(size)…);

Create table table\_name **as** select column\_name1, column\_name2,…colmn\_nameN from table\_name where predicate;

**AS Subquery**

Subquery is the select statement that defines the set of rows to be inserted into the new table.

**Example**

Create table dept80 as select employee\_id, last\_name, salary\*12 Annsal, hire\_date from employees where dept\_id=80;

**The ALTER TABLE Statement**

The ALTER statement is used to

* Add a new column
* Modify an existing column
* Define a default value to the new column
* Drop a column
* To include or drop integrity constraint.

**SYNTAX**

ALTER TABLE table\_name ADD /MODIFY(Column\_name type(size));

ALTER TABLE table\_name DROP COLUMN (Column\_nname);

ALTER TABLE ADD CONSTRAINT Constraint\_name PRIMARY KEY (Colum\_Name);

**Example:**

Alter table dept80 add (jod\_id varchar2(9));

Alter table dept80 modify (last\_name varchar2(30));

Alter table dept80 drop column job\_id;

***NOTE*:** *Once the column is dropped it cannot be recovered.*

**DROPPING A TABLE**

* All data and structure in the table is deleted.
* Any pending transactions are committed.
* All indexes are dropped.
* Cannot roll back the drop table statement.

**Syntax:**

**Drop table *tablename;***

**Example:**

Drop table dept80;

**RENAMING A TABLE**

To rename a table or view.

**Syntax**

RENAME old\_name to new\_name

**Example:**

Rename dept to detail\_dept;

**TRUNCATING A TABLE**

Removes all rows from the table.

Releases the storage space used by that table.

**Syntax**

TRUNCATE TABLE *table\_name*;

E**xample:**

TRUNCATE TABLE copy\_emp;

**Self-Activity 1:**

1. Create a table with Serial No, Country Name and Continent in a table called Country. Then rename the table to my\_country.

|  |  |  |
| --- | --- | --- |
| S\_No | Country Name | Continent |
| 1 | India | Asia |
| 2 | United States | North America |
| 3 | Egypt | Africa |

|  |
| --- |
| **Ans:** |
|  |
| **Output:** |

2. Add a column Language in the table my\_country and populate the column

|  |  |  |  |
| --- | --- | --- | --- |
| S\_No | Country Name | Continent | Language |
| 1 | India | Asia | Hindi |
| 2 | United States | North America | English |
| 3 | Egypt | Africa | Egyptian |

|  |
| --- |
| **Ans:** |
|  |
| **Output:** |

3. Change the data type of S.No column to FLOAT

|  |  |  |  |
| --- | --- | --- | --- |
| S\_No | Country Name | Continent | Language |
| 1.0 | India | Asia | Hindi |
| 2.0 | United States | North America | English |
| 3.0 | Egypt | Africa | Egyptian |

|  |
| --- |
| **Ans:** |
|  |
| **Output:** |

4. Now drop the column S\_No from my\_country table.

|  |  |  |
| --- | --- | --- |
| Country Name | Continent | Language |
| India | Asia | Hindi |
| United States | North America | English |
| Egypt | Africa | Egyptian |

|  |
| --- |
| **Ans:** |
|  |
| **Output:** |

5. Drop the Language column and add the column Population after Country\_Name of the type INT.

|  |  |  |
| --- | --- | --- |
| Country Name | Population( in billion) | Language |
| India | 140 | Hindi |
| United States | 50 | English |
| Egypt | 20 | Egyptian |

Then change the column name CountryName to Country of varchar(25) specification.

|  |
| --- |
| **Ans:** |
|  |
| **Output:** |

**Viva Questions:**

1. Differentiate between Truncate and Drop. Can truncate and Drop be rolled back? why or why not?

Ans: Truncate: Truncate is used to delete all rows in a table/relation resulting in an empty table.

Drop: Drop is used to delete the entire table, as a result the entire table does not exist after drop is used.

‘Truncate’ can be rolled back whereas ‘Drop’ cannot be rolled back. This is because ‘Drop’ is a DDL command that is committed automatically once executed, meaning that the change is permanent and irreversible whereas ‘Truncate’ is not committed until the ‘commit’ query is executed.

1. What will be the result if you try to drop a column involving composite primary key?

Ans: When a column involving a composite primary key is dropped it will result in an error that does not allow that column to be dropped. This is because the primary key constraint ensures that the composite columns cannot be removed individually. In order to drop the CColumn the primary key must be dropped first.

1. Write the purpose of NVL function in SQL.

Ans: The purpose of NVL function in SQL is to replace the NULL values in a column with an actual meaningful value.

Syntax: NVL(<column name to check NULL>, <value to replace with NULL>)

Example: select NVL(email, "Not Available") from employees;

1. How does altering a table (adding/dropping columns) affect existing indexes and constraints?

Ans: Altering a table by adding does not affect the existing indexes or constraints.

Altering a table by dropping a column that is a part of an index or a constraint like a primary key or foreign key is not allowed and will result in an error unless the constraints are dropped first.

1. When you rename a table or column, how can you ensure all dependent objects like views and procedures are automatically updated?

Ans: In order to ensure that all dependent objects are updated automatically, IDEs like Oracle SQL can be used. They detect name changes within a database and automatically update dependent objects.

**Assignment Evaluation:**

**0: Not Done [ ] 1: Incomplete [ ]**

**2: Late Complete [ ] 3: Needs Improvement [ ]**

**4: Complete [ ] 5: Well Done [ ]**

**Activity Sheet 2: Date:**

**DML COMMANDS - MANIPULATING DATA**

**OBJECTIVE**

After, the completion of this exercise the students will be able to do the following

* Describe each DML statement
* Insert rows into tables
* Update rows into table
* Delete rows from table
* Control Transactions

A DML statement is executed when you:

* Add new rows to a table
* Modify existing rows
* Removing existing rows

A transaction consists of a collection of DML statements that form a logical unit of work.

**To Add a New Row**

INSERT Statement

**Syntax**

INSERT INTO table\_name VALUES (column1 values, column2 values, …, columnn values);

**Example:**

INSERT INTO department (70, ‘Public relations’, 100,1700);

**Inserting rows with null values**

**Implicit Method:** (Omit the column)

INSERT INTO department VALUES (30,’purchasing’);

**Explicit Method:** (Specify NULL keyword)

INSERT INTO department VALUES (100,’finance’, NULL, NULL);

**Inserting Special Values**

**Example:**

Using SYSDATE INSERT INTO employees VALUES (113,’louis’, ‘popp’, ‘lpopp’,’5151244567’,**SYSDATE**, ‘ac\_account’, 6900, NULL, 205, 100);

**Inserting Specific Date Values**

**Example:**

INSERT INTO employees VALUES ( 114,’den’, ‘raphealy’, ‘drapheal’, ‘5151274561’, **TO\_DATE(‘feb 3,1999’,’mon, dd ,yyyy’),** ‘ac\_account’, 11000,100,30);

**To Insert Multiple Rows**

**&** is the placeholder for the variable value

**Example:**

INSERT INTO department VALUES (ept\_id1, dept\_name1, location1), (ept\_id2, dept\_name2, location2),…;

**Copying Rows from another table**

* Using Subquery

**Example:**

INSER INTO sales\_reps(id, name, salary, commission\_pct)

SELECT employee\_id, Last\_name, salary, commission\_pct

FROM employees

WHERE jod\_id LIKE ‘%REP’);

**CHANGING DATA IN A TABLE**

UPDATE Statement

**Syntax1: (** to update specific rows**)**

UPDATE table\_name SET column=value WHERE condition;

**Syntax 2:** (To updae all rows)

UPDATE table\_name SET column=value;

**Updating columns with a subquery**

UPDATE employees SET job\_id= (SELECT job\_id FROM employees WHERE employee\_id=205) WHERE employee\_id=114;

**REMOVING A ROW FROM A TABLE**

**DELETE STATEMENT**

**Syntax**

DELETE FROM table\_name WHERE conditions;

**Example:**

DELETE FROM department WHERE dept\_name=’finance’’;

**Capabilities of SQL SELECT statement**

A SELECT statement retrieves information from the database. Using a select statement, we can perform

* Projection: To choose the columns in a table
* Selection: To choose the rows in a table
* Joining: To bring together the data that is stored in different tables

**Basic SELECT Statement**

**Syntax**

SELECT \*|DISTINCT Column\_ name| alias FROM table\_name;

**NOTE:**

DISTINCT—Suppress the duplicates.

Alias—gives selected columns different headings.

**Example: 1**

SELECT \* FROM departments;

**Example: 2**

SELECT location\_id, department\_id FROM departments;

**Writing SQL Statements**

* SQL statements are not case sensitive
* SQL statements can be on one or more lines.
* Keywords cannot be abbreviated or split across lines
* Clauses are usually placed on separate lines
* Indents are sued to enhance readability

**Limiting the Rows selected**

* Using WHERE clause
* Alias cannot used in WHERE clause

**Syntax**

SELECT----------

FROM----------

WHERE condition;

**Example:**

SELECT employee\_id,last\_name, job\_id, deparment\_id FROM employees WHERE department\_id=90;

**Character strings and Dates**

Character strings and date values are enclosed in single quotation marks.

Character values are case sensitive and date values are format sensitive.

**Example:**

SELECT employee\_id,last\_name, job\_id, deparment\_id FROM employees

WHERE last\_name=’WHALEN”;

**Self-Activity 2:**

1. Write a SQL statement to change the email column of employees table with 'not available' for all employees.

**Note: Creating table and Inserting Values to be done before starting the exercise.**

Here are the sample table employees.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| EMPLOYEE\_ID | FIRST\_NAME | LAST\_NAME | EMAIL | PHONE\_NUMBER | HIRE\_DATE | JOB\_ID | SALARY | COMMISSION\_PCT | MANAGER\_ID | DEPARTMENT\_ID |
| 100 | Steven | King | SKING | 515.123.4567 | 1987-06-17 | AD\_PRES | 24000.00 | 0.00 | 0 | 90 |
| 101 | Neena | Kochhar | NKOCHHAR | 515.123.4568 | 1987-06-18 | AD\_VP | 17000.00 | 0.00 | 100 | 90 |
| 102 | Lex | De Haan | LDEHAAN | 515.123.4569 | 1987-06-19 | AD\_VP | 17000.00 | 0.00 | 100 | 90 |
| 103 | Alexander | Hunold | AHUNOLD | 590.423.4567 | 1987-06-20 | IT\_PROG | 9000.00 | 0.00 | 102 | 60 |
| 104 | Bruce | Ernst | BERNST | 590.423.4568 | 1987-06-21 | IT\_PROG | 6000.00 | 0.00 | 103 | 60 |

|  |
| --- |
| **Ans:** |
|  |
| **Output:** |

1. Write a SQL statement to change the email and commission\_pct column of employees table with 'not available' and 0.10 for those employees whose department\_id is 60.

|  |
| --- |
| **Ans:** |
|  |
| **Output:** |

1. Write a SQL statement to change the email column of employees table with 'not available' for those employees whose department\_id is 60 and gets a commission\_pct is less than .20

|  |
| --- |
| **Ans:** |
|  |
| **Output:** |

1. Write a SQL statement to delete the employee details whose Job ID is “ AD\_VP” and Department ID is 90.

|  |
| --- |
| **Ans:** |
|  |
| **Output:** |

**Consider the following tables Table: Salesman**

|  |  |  |  |
| --- | --- | --- | --- |
| **Salesman\_id** | **Name** | **City** | **Commisision** |
| 5001 | James Hoog | Newyork | 0.15 |
| 5002 | Nail Knite | Paris | 0.13 |
| 5005 | Pit Alex | London | 0.11 |
| 5006 | Mc Lyon | Paris | 0.14 |
| 5003 | Lauson Hen |  | 0.12 |
| 5007 | Paul Adam | Rome | 0.13 |

**Table : Customer**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Customer\_id** | **Customer\_name** | **City** | **Grade** | **Salesman\_id** |
| 3002 | Nick Rimando | Newyork | 100 | 5001 |
| 3005 | Graham Lusi | California | 200 | 5002 |
| 3001 | Brad Guran | London |  |  |
| 3004 | Fabian Johns | Paris | 300 | 5006 |
| 3007 | Brad Davis | Newyork | 200 | 5001 |
| 3009 | Geoff Camero | Berlin | 100 |  |
| 3008 | Julian Green | London | 300 | 5002 |
| 3003 | Jory Altidor | Moncow | 200 | 5007 |

Table:Order

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Order\_no** | **Purch\_amt** | | **Order\_date** | **Customer\_id** | **Salesman\_id** |
| 70001 | | 150.5 | 2016-10-05 | 3005 | 5002 |
| 70009 | | 270.65 | 2016-09-10 | 3001 |  |
| 70002 | | 65.26 | 2016-10-05 | 3002 | 5001 |
| 70004 | | 110.5 | 2016-08-17 | 3009 |  |
| 70007 | | 948.5 | 2016-09-10 | 3005 | 5002 |
| 70005 | | 2400.6 | 2016-07-27 | 3007 | 5001 |
| 70008 | | 5760 | 2016-09-10 | 3002 | 5001 |
| 70010 | | 1983.43 | 2016-10-10 | 3004 | 5006 |
| 70003 | | 2480.4 | 2016-10-10 | 3009 |  |
| 70012 | | 250.45 | 2016-06-27 | 3008 | 5002 |
| 70011 | | 75.29 | 2016-08-17 | 3003 | 5007 |

**Write the below statements in SQL**

|  |
| --- |
| 1. **Display name and commission of all salesman.** |
| **Query :** |
| **Output:** |

|  |  |
| --- | --- |
| **2.** **Retrieve salesman id of all salesmen from orders table without any repeats**. | |
| **Query :** | |
| **Output:** | |
| 1. **Display names and city of salesman, who belongs to the city of Paris.** |
| **Query :** |
| **Output:** |

|  |
| --- |
| 1. **Display all the information for those customers with a grade of 200**. |
| **Query:** |
| **Output:** |

|  |
| --- |
| 1. **Display the order number, order date and the purchase amount for orders which will be delivered by the salesman with ID 5001.** |
| **Query :** |
| **Output:** |

|  |
| --- |
| 1. **Show the winner of the 1971 prize for Literature.** |
| **Query:** |
| **Output:** |

|  |
| --- |
| 1. **Show all the details of the winners with first name Louis.** |
| **Query :** |
| **Output:** |

|  |
| --- |
| 1. **Show all the winners in Physics for 1970 together with the winner of Economics for 1971** |
| **Query :** |
| **Output:** |

|  |
| --- |
| 1. **Show all the winners of Nobel prize in the year 1970 except the subject Physiology and Economics.** |
| **Query :** |
| **Output:** |

|  |
| --- |
| 1. **Find all the details of the Nobel winners for the subject not started with the letter 'P' and arranged the list as the most recent comes first, then by name in order** |
| **Query :** |
| **Output:** |

|  |
| --- |
| 1. **Find the name and price of the cheapest item(s).** |
| **Query :** |
| **Output:** |

|  |
| --- |
| 1. **Display all the customers, who are either belongs to the city New York or not had a grade above 100.** |
| **Query :** |
| **Output:** |

|  |
| --- |
| 1. **Find those salesmen with all information who gets the commission within a range of 0.12 and 0.14** |
| **Query :** |
| **Output:** |

|  |
| --- |
| 1. **Find all those customers with all information whose names are ending with the letter 'n'** |
| **Query :** |
| **Output:** |
| 1. **Find those salesmen with all information whose name containing the 1st character is 'N' and the 4th character is 'l' and rests may be any character..** |
| **Query :** |
| **Output:** |

|  |
| --- |
| 1. **Find that customer with all information who does not get any grade except NULL.** |
| **Query :** |
| **Output:** |

|  |
| --- |
| 1. **Find the total purchase amount of all orders.** |
| **Query :** |
| **Output:** |

|  |
| --- |
| **18.** **Find the number of salesman currently listing for all of their customers**. |
| **Query :** |
| **Output:** |

|  |
| --- |
| **19. Find the highest grade for each of the cities of the customers.** |
| **Query :** |
| **Output:** |

|  |
| --- |
| **20.** **Find the highest purchase amount ordered by each customer with their ID and highest purchase amount**. |
| **Query:** |
| **Output:** |

|  |
| --- |
| **21.** **Find the highest purchase amount ordered by each customer on a particular date with their ID, order date and highest purchase amount**. |
| **Query:** |
| **Output:** |

|  |
| --- |
| **22.** **Find the highest purchase amount on a date '2012-08-17' for each salesman with their ID**. |
| **Query :** |
| **Output:** |

|  |
| --- |
| **23.** **Find the highest purchase amount with their customer ID and order date, for only those customers who have the highest purchase amount in a day is more than 2000.** |
| **Query :** |
| **Output:** |

|  |
| --- |
| **24.** **Write a SQL statement that counts all orders for a date August 17th, 2012**. |
| **Query :** |
| **Output:** |

**Viva Questions:**

1. What happens if you try to insert a row without specifying a value for a NOT NULL column that has no default? How does this differ between databases like MySQL and PostgreSQL?

ANS :

If you try to insert a row without specifying a value for a NOT NULL column that has no default, the database will reject the operation with an error. In PostgreSQL, this always results in an error, strictly enforcing the NOT NULL constraint. In MySQL, behavior depends on the SQL mode: if strict mode (like STRICT\_TRANS\_TABLES) is enabled, it throws an error similar to PostgreSQL; however, if strict mode is disabled, MySQL may silently insert an implicit default value such as 0 or an empty string, which can lead to unintended data being stored.

1. How would you update a table based on a condition from a different table (e.g., update salaries in employees based on salary grades)? How would you do it without using JOINs?

ANS:

UPDATE employees SET salary = salary + (SELECT increment FROM salary\_grades WHERE employees.salary BETWEEN min\_salary AND max\_salary);

1. Is there a performance difference between DELETE and TRUNCATE? When would you prefer one over the other, and what are the risks?

ANS:

Yes, there is a performance difference between DELETE and TRUNCATE: TRUNCATE is generally much faster because it deallocates all data pages in the table without scanning individual rows, while DELETE removes rows one by one and logs each deletion, which is slower and generates more transaction log entries. You would prefer TRUNCATE when you need to quickly remove all rows from a table and do not require per-row delete triggers or fine-grained logging. However, TRUNCATE cannot be used if there are foreign key constraints referencing the table, does not fire ON DELETE triggers, and often cannot be rolled back in the same way as DELETE in some databases, so you should use it cautiously when transactional consistency or referential integrity is important.

1. How can subqueries in the SELECT clause (scalar subqueries) be optimized, especially when dealing with large data sets?

ANS:

Scalar subqueries in the SELECT clause can be optimized by rewriting them as JOINs or APPLY (in SQL Server), using indexing on the subquery’s filtering columns, and ensuring the subquery is not re-evaluated for every row.

1. You need to delete records older than 10 years from a large audit log table. What strategies would you use to avoid locking issues and performance degradation?

ANS:

To delete records older than 10 years from a large audit log table without causing locking issues or performance degradation, you can delete in small batches using a loop or script that repeatedly issues DELETE statements with a LIMIT clause (e.g., DELETE FROM audit\_log WHERE created\_at < NOW() - INTERVAL '10 years' LIMIT 1000) to avoid long-running transactions and reduce locking contention; additionally, you can schedule these deletes during low-traffic periods, disable or defer indexes and constraints if possible, and consider partitioning the table by date so that dropping an old partition instantly removes historical data with minimal impact on performance.

**Assignment Evaluation:**

**0: Not Done [ ] 1: Incomplete [ ]**

**2: Late Complete [ ] 3: Needs Improvement [ ]**

**4: Complete [ ] 5: Well Done [ ]**

**Activity Sheet3: Date:**

**SQL CONSTRAINTS**

**OBJECTIVE**

After the completion of this exercise the students should be able to do the following

* Describe the constraints
* Create and maintain the constraints

**What are Integrity Constraints?**

* Constraints enforce rules at the table level.
* Constraints prevent the deletion of a table if there are dependencies

**The following types of integrity constraints are valid**

1. **Domain Integrity**

* NOT NULL
* CHECK

1. **Entity Integrity**

* UNIQUE
* PRIMARY KEY

1. **Referential Integrity**

* FOREIGN KEY

**Constraints can be created in either of two ways**

1. At the same time as the table is created
2. After the table has been created.

**Defining Constraints**

Create table tablename (column\_name1 data\_ type constraints, column\_name2 data\_ type constraints …);

**Example:**

Create table employlees ( employee\_id number(6), first\_name varchar2(20), ..job\_id varchar2 (10), CONSTRAINT emp\_emp\_id\_pk PRIMARY KEY (employlee\_id));

**Domain Integrity**

This constraint sets a range and any violations that takes place will prevent the user from performing the manipulation that caused the breach. It includes:

**NOT NULL Constraint**

While creating tables, by default the rows can have null value. The enforcement of not null constraint in a table ensure that the table contains values.

**Principle of null values:**

* Setting null value is appropriate when the actual value is unknown, or when a value would not be meaningful.
* A null value is not equivalent to a value of zero.
* A null value will always evaluate to null in any expression.
* When a column name is defined as not null, that column becomes a mandatory i.e., the user has to enter data into it.
* Not null Integrity constraint cannot be defined using the alter table command when the table contain rows.

**Example**

CREATE TABLE employees (employee\_id number (6), last\_name varchar2(25) NOT NULL, salary number(8,2), commission\_pct number(2,2), hire\_date date constraint emp\_hire\_date\_nn NOT NULL’….);

**CHECK**

Check constraint can be defined to allow only a particular range of values, when the manipulation violates this constraint,the record will be rejected.Check condition cannot contain sub queries.

CREATE TABLE employees (employee\_id number (6), last\_name varchar2 (25) NOT NULL, salary number(8,2), commission\_pct number(2,2), hire\_date date constraint emp\_hire\_date\_nn NOT NULL’…,CONSTRAINT emp\_salary\_mi CHECK(salary > 0));

**Entity Integrity**

Maintains uniqueness in a record. An entity represents a table and each row of a table represents an instance of that entity. To identify each row in a table uniquely we need to use this constraint. There are 2 entity constraints:

**a) Unique key constraint**

It is used to ensure that information in the column for each record is unique, as with telephone or driver’s license numbers. It prevents the duplication of value with rows of a specified column in a set of column. A column defined with the constraint can allow null value.

If unique key constraint is defined in more than one column i.e., combination of column cannot be specified. Maximum combination of columns that a composite unique key can contain is 16.

**Example:**

CREATE TABLE employees (employee\_id number(6), last\_name varchar2(25) NOT NULL,email varchar2(25), salary number(8,2), commission\_pct number(2,2), hire\_date date constraint emp\_hire\_date\_nn NOT NULL’ COSTRAINT emp\_email\_uk UNIQUE(email));

**PRIMARY KEY CONSTRAINT**

A primary key avoids duplication of rows and does not allow null values. Can be defined on one or more columns in a table and is used to uniquely identify each row in a table. These values should never be changed and should never be null.

A table should have only one primary key. If a primary key constraint is assigned to more than one column or combination of column is said to be composite primary key, which can contain 16 columns

**Example:**

CREATE TABLE employees (employee\_id number(6) , last\_name varchar2(25) NOT NULL,email varchar2(25), salary number(8,2), commission\_pct number(2,2), hire\_date date constraint emp\_hire\_date\_nn NOT NULL, Constraint emp\_id pk PRIMARY KEY (employee\_id),CONSTRAINT emp\_email\_uk UNIQUE(email));

1. **Referential Integrity**

It enforces relationship between tables. To establish parent-child relationship between 2 tables having a common column definition, we make use of this constraint. To implement this, we should define the column in the parent table as primary key and same column in the child table as foreign key referring to the corresponding parent entry.

**Foreign key**

A column or combination of column included in the definition of referential integrity, which would refer to a referenced key.

**Referenced key**

It is a unique or primary key upon which is defined on a column belonging to the parent table.

Keywords:

**FOREIGN KEY:** Defines the column in the child table at the table level constraint.

**REFERENCES:** Identifies the table and column in the parent table.

**ON DELETE CASCADE:** Deletes the dependent rows in the child table when a row in the parent table is deleted.

**ON DELETE SET NULL:** converts dependent foreign key values to null when the parent value is removed.

CREATE TABLE employees (employee\_id number(6) , last\_name varchar2(25) NOT NULL,email varchar2(25), salary number(8,2), commission\_pct number(2,2), hire\_date date constraint emp\_hire\_date\_nn NOT NULL, Constraint emp\_id pk PRIMARY KEY (employee\_id),CONSTRAINT emp\_email\_uk UNIQUE(email),CONSTRAINT emp\_dept\_fk FOREIGN KEY (department\_id) references deparments(dept\_id));

**ADDING A CONSTRAINT**

Use the ALTER to

* Add or Drop a constraint, but not modify the structure
* Enable or Disable the constraints
* Add a not null constraint by using the Modify clause

**Syntax**

ALTER TABLE table\_name ADD CONSTRAINT Cons\_name type(column name);

**Example:**

ALTER TABLE employees ADD CONSTRAINT emp\_manager\_fk FOREIGN KEY (manager\_id) REFERENCES employees (employee\_id);

**DROPPING A CONSTRAINT**

**Example:**

ALTER TABLE employees DROP CONSTRAINT emp\_manager\_fk;

**CASCADE IN DROP**

* The CASCADE option of the DROP clause causes any dependent constraints also to be dropped.

**Syntax**

ALTER TABLE departments DROP PRIMARY KEY|UNIQUE (column)| CONSTRAINT constraint \_name CASCADE;

**DISABLING CONSTRAINTS**

* Execute the DISABLE clause of the ALTER TABLE statement to deactivate an integrity constraint
* Apply the CASCADE option to disable dependent integrity constraints.

**Example**

ALTER TABLE employees DISABLE CONSTRAINT emp\_emp\_id\_pk CASCADE;

**ENABLING CONSTRAINTS**

* Activate an integrity constraint currently disabled in the table definition by using the ENABLE clause.

**Example**

ALTER TABLE employees ENABLE CONSTRAINT emp\_emp\_id\_pk CASCADE;

**CASCADING CONSTRAINTS**

The CASCADE CONSTRAINTS clause is used along with the DROP column clause.

It drops all referential integrity constraints that refer to the primary and unique keys defined on the dropped Columns.

This clause also drops all multicolumn constraints defined on the dropped column.

**Example:**

**Assume table TEST1 with the following structure**

CREATE TABLE test1 ( pk number PRIMARY KEY, fk number, col1 number,col2 number, CONTRAINT fk\_constraint FOREIGN KEY(fk) references test1, CONSTRAINT ck1 CHECK (pk>0 and col1>0), CONSTRAINT ck2 CHECK (col2>0));

**An error is returned for the following statements**

ALTER TABLE test1 DROP (pk);

ALTER TABLE test1 DROP (col1);

**The above statement can be written with CASCADE CONSTRAINT**

ALTER TABLE test 1 DROP(pk) CASCADE CONSTRAINTS;

**(OR)**

ALTER TABLE test 1 DROP(pk, fk, col1) CASCADE CONSTRAINTS;

**VIEWING CONSTRAINTS**

Query the USER\_CONSTRAINTS table to view all the constraints definition and names.

**Example:**

SELECT constraint\_name, constraint\_type, search\_condition FROM user\_constraints

WHERE table\_name=’employees’;

**Viewing the columns associated with constraints**

SELECT constraint\_name, constraint\_type, FROM user\_cons\_columns

WHERE table\_name=’employees’;

**Self-Activity 3:**

1. Create a table called Mytable ; “UserID” a integer, “FirstName” a string, “LastName” a string columns will NOT accept NULL values when the “Mytable” table is created. But JobPosition column which is a string does not have any constraint.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Now alter the table to make JobPosition also have NOT NULL constraint

|  |
| --- |
| **Ans:** |
| **Query :** |
| **put:** |

1. Now alter the same table to make UserID UNIQUE

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Now remove the constraint on UserID. Make a constraint such that UNIQUE constraint is added on two columns as UC\_MyTab on UserID and JobPosition

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Now remove the above constraint. Make a constraint PK\_Mytab as Primary Key on User\_ID and Position.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Create tables as below and materialize the foreign key constraint There are two tables named **Customer Table**

CREATE TABLE Customer (

CustomerID int NOT NULL PRIMARY KEY, Name varchar(45) NOT NULL,

Age int,

City varchar(25)

);

**Orders Table**

CREATE TABLE Orders (

Order\_ID int NOT NULL PRIMARY KEY,

Order\_Num int NOT NULL, CustomerID int,

FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID)

);

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Now add VoterID in MyTable table as a integer. Check whether it is greater than 18 using ALTER command.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. How would you assign a default value for JobPosition in MyTable as “Technical”

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Make the UserID as autoincrement and insert three rows in the table.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Create a Mov Table on Movie\_Cassette Library as per below specification.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **No.** | **Title** | **Type** | **Rating** | **Stars** | **Qty** | **Price** |
| 1 | Gone with the Wind | Drama | G | Gable | 4 | 39.95 |
| 2 | Friday the 13th | Horror | R | Jason | 2 | 69.95 |
| 3 | Top Gun | Drama | PG | Cruise | 7 | 49.95 |
| 4 | Splash | Comedy | PG13 | Hanks | 3 | 29.95 |
| 5 | Independence Day | Drama | R | Turner | 3 | 19.95 |
| 6 | Risky Business | Comedy | R | Cruise | 2 | 44.95 |
| 7 | Cocoon | Scifi | PG | Ameche | 2 | 31.95 |
| 8 | Crocodile Dundee | Comedy | PG13 | Harris | 2 | 69.95 |
| 9 | 101 Dalmatians | Comedy | G |  | 3 | 59.95 |
| 10 | Tootsie | Comedy | PG | Hoffman | 1 | 29.95 |

* 1. Find the total value of the movie cassettes available in the library.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

* 1. Display a list of all movies with Price over 20 and sorted by Price.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

* 1. Display all the movies sorted by Qty in decreasing order.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

10.4 Display a report listing a movie number, current value and replacement value for each movie in the above

table. Calculate the eplacement value for all movies as QTY \* Price \* 1.15

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

* 1. Count the number of movies where Rating is not “G”.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

* 1. Insert a new movie in MOV table. Fill all the columns with values.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Create a table called DOCTOR and SALARY as below and execute the query below. DOCTOR

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **DEPT** | **SE**  **X** | **EXPERIE**  **NCE** |
| 101 | John | ENT | M | 12 |
| 104 | Smith | ORTHOPE DIC | M | 5 |
| 107 | George | CARDIOL  OGY | M | 10 |
| 114 | Lara | SKIN | F | 3 |
| 109 | K George | MEDICINE | F | 9 |
| 105 | Johnson | ORTHOPE  DIC | M | 10 |
| 117 | Luc  y | ENT | F | 3 |
| 118 | Bill  y | MEDICINE | F | 12 |
|  | Morphy | ORTHOPE  DIC | M | 15 |

|  |  |  |  |
| --- | --- | --- | --- |
| **SALARY 1D** | **BASIC** | **ALLOWA NCE** | **CONSULTAT ION** |
| 101 | 12000 | 1000 | 300 |
| 104 | 23000 | 2300 | 500 |
| 107 | 32000 | 4000 | 500 |
| 114 | 12000 | 5200 | 100 |
| 109 | 42000 | 1700 | 200 |
| 105 | 18900 | 1690 | 300 |
| 130 | 21700 | 2600 | 300 |

11.1 Display NAME of all doctors who are in “MEDICINE” having more than 10 year experience from the

table DOCTOR.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

* 1. Display the average salary of all doctors working in “ENT” department using the tables DOCTOR and SALARY. Salary=BASIC + ALLOWANCE.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

* 1. Display the minimum ALLOWANCE of female doctors.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

* 1. Display the highest consultation fee among all male doctor.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

* 1. Write a query to Groupby sex and find count of male and female doctors

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

* 1. Write a query to Groupby sex and find average salary

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

**Viva Questions:**

1. Which constraint ensures that a column cannot have NULL values?
2. Why is it important to define constraints like NOT NULL, UNIQUE, and CHECK at the time of table creation rather than adding them later?
3. What are the potential downsides of using too many constraints or indexes in a frequently updated table?
4. How can you make a column auto-increment in SQL?
5. How do you assign a default value to a column?

**Assignment Evaluation:**

0: Not Done [ ] 1: Incomplete [ ]

2: Late Complete [ ] 3: Needs Improvement [ ]

4: Complete [ ] 5: Well Done [ ]

**ACTIVITY SHEET 4: DATE:**

**IMPLEMENTATION OF ARITHMETIC / LOGIC OPERATIONS, SORTING AND GROUPING**

**OBJECTIVES**

After the completion of this exercise, the students will be able to do the following:

* List the capabilities of SQL SELECT Statement
* Execute a basic SELECT statement using arithmetic operators.

**Using Arithmetic Expressions**

Basic Arithmetic operators like \*, /, +, -can be used

**Example:1**

SELECT last\_name, salary, salary+300 FROM employees;

**Example:2**

SELECT last\_name, salary, 12\*salary+100 FROM employees;

The statement is not same as

SELECT last\_name, salary, 12\*(salary+100) FROM employees;

**Example:3**

SELECT last\_name, job\_id, salary, commission\_pct FROM employees;

**Example:4**

SELECT last\_name, job\_id, salary, 12\*salary\*commission\_pct FROM employees;

**Using Column Alias**

* To rename a column heading with or without AS keyword.

**Example:1**

SELECT last\_name AS Name

FROM employees;

**Example: 2**

SELECT last\_name “Name” salary\*12 “Annual Salary “

FROM employees;

**Concatenation Operator**

* Concatenates columns or character strings to other columns
* Represented by two vertical bars (||)
* Creates a resultant column that is a character expression

**Example:**

SELECT last\_name||job\_id AS “EMPLOYEES JOB” FROM employees;

**Using Literal Character String**

* A literal is a character, a number, or a date included in the SELECT list.
* Date and character literal values must be enclosed within single quotation marks.

**Example:**

SELECT last\_name||’is a’||job\_id AS “EMPLOYEES JOB” FROM employees;

**Eliminating Duplicate Rows**

* Using DISTINCT keyword.

**Example:**

SELECT DISTINCT deparment\_id FROM employees

**Displaying Table Structure**

* Using DESC keyword.

**Syntax**

DESC table\_name;

**Example:**

DESC employees;

**Comparison Conditions**

All relational operators can be used. (=, >, >=, <, <= ,<>,!=)

**Example:**

SELECT last\_name, salary

FROM employees

WHERE salary<=3000;

**Other comparison conditions**

|  |  |
| --- | --- |
| Operator | Meaning |
| BETWEEN  …AND… | Between two values |
| IN | Match any of a list of values |
| LIKE | Match a character pattern |
| IS NULL | Is a null values |

**Example:1**

SELECT last\_name, salary

FROM employees

WHERE salary BETWEEN 2500 AND 3500;

**Example:2**

SELECT employee\_id, last\_name, salary , manager\_id

FROM employees

WHERE manager\_id IN (101, 100,201);

**Example:3**

* Use the LIKE condition to perform wildcard searches of valid string values.
* Two symbols can be used to construct the search string
* % denotes zero or more characters
* \_ denotes one character

SELECT first\_name, salary

FROM employees

WHERE first\_name LIKE ‘%s’;

**Example:4**

SELECT last\_name, salary

FROM employees

WHERE last\_name LIKE ‘\_o%’;

**Example:5**

**ESCAPE option**-To have an exact match for the actual % and\_ characters

To search for the string that contain ‘SA\_’

SELECT employee\_id, first\_name, salary,job\_id

FROM employees

WHERE job\_id LIKE ‘%sa\\_%’ESCAPE’\’;

**Test for NULL**

* Using IS NULL operator

**Example:**

SELECT employee\_id, last\_name, salary , manager\_id

FROM employees

WHERE manager\_id IS NULL;

**Logical Conditions**

All logical operators can be used.( AND,OR,NOT)

**Example:1**

SELECT employee\_id, last\_name, salary , job\_id

FROM employees

WHERE salary>=10000

AND job\_id LIKE ‘%MAN%’;

**Example:2**

SELECT employee\_id, last\_name, salary , job\_id

FROM employees

WHERE salary>=10000

OR job\_id LIKE ‘%MAN%’;

**Example:3**

SELECT employee\_id, last\_name, salary , job\_id

FROM employees WHERE job\_id NOT IN (‘it\_prog’, st\_clerk’, sa\_rep’);

**Rules of Precedence**

|  |  |
| --- | --- |
| **Order Evaluated** | **Operator** |
| 1 | Arithmetic |
| 2 | Concatenation |
| 3 | Comparison |
| 4 | IS [NOT] NULL, LIKE, [NOT] IN |
| 5 | [NOT] BETWEEN |
| 6 | Logical NOT |
| 7 | Logical AND |
| 8 | Logical OR |

**Example:1**

SELECT employee\_id, last\_name, salary , job\_id

FROM employees

WHERE job\_id =’sa\_rep’

OR job\_id=’ad\_pres’

AND salary>15000;

**Example:2**

SELECT employee\_id, last\_name, salary , job\_id

FROM employees

WHERE (job\_id =’sa\_rep’

OR job\_id=’ad\_pres’)

AND salary>15000;

**Sorting the rows**

Using ORDER BY Clause

**ASC**-Ascending Order,Default

**DESC**-Descending order

**Example:1**

SELECT last\_name, salary , job\_id,department\_id,hire\_date

FROM employees

ORDER BY hire\_date;

**Example:2**

SELECT last\_name, salary , job\_id,department\_id,hire\_date

FROM employees

ORDER BY hire\_date DESC;

**Example:3**

**Sorting by column alias**

SELECT last\_name, salary\*12 annsal , job\_id,department\_id,hire\_date

FROM employees

ORDER BY annsal;

**Example:4**

**Sorting by Multiple columns**

SELECT last\_name, salary , job\_id,department\_id,hire\_date

FROM employees

ORDER BY department\_id, salary DESC;

**Self-Activity 4:**

Consider the Employee table and solve the following queries:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| EMPLOYEE\_ID | FIRST\_NAME | LAST\_NAME | EMAIL | PHONE\_NUMBER | HIRE\_DATE | JOB\_ID | SALARY | COMMISSION\_PCT | MANAGER\_ID | DEPARTMENT\_ID |
| 100 | Steven | King | SKING | 515.123.4567 | 1987-06-17 | AD\_PRES | 24000.00 | 0.00 | 0 | 90 |
| 101 | Neena | Kochhar | NKOCHHAR | 515.123.4568 | 1987-06-18 | AD\_VP | 17000.00 | 0.00 | 100 | 90 |
| 102 | Lex | De Haan | LDEHAAN | 515.123.4569 | 1987-06-19 | AD\_VP | 17000.00 | 0.00 | 100 | 90 |
| 103 | Alexander | Hunold | AHUNOLD | 590.423.4567 | 1987-06-20 | IT\_PROG | 9000.00 | 0.00 | 102 | 60 |
| 104 | Bruce | Ernst | BERNST | 590.423.4568 | 1987-06-21 | IT\_PROG | 6000.00 | 0.00 | 103 | 60 |

1. Write a query to display the name (first\_name, last\_name) and salary for all employees whose salary is not in the range $10,000 through $15,000.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a query to display the name (first\_name, last\_name) and department ID of all employees in departments 30 or 100 in ascending order.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a query to display the first\_name of all employees who have both "e" and "r" in their first name.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a query to display the last name of employees whose names have exactly 6 characters.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a query to select all record from employees where last name in 'KING' and 'ERNST'.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Display the last name and hire date of every employee who was hired in 1987. (hints: like)

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a query to select rows ordered by DEPARTMENT\_ID

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Display the last name and Job ID of all employees who do not have a manager. (hints: is null)

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Display the last name, salary, and commission for all employees who earn commissions. Sort data in descending order of salary and commissions.(hints: is not nul,orderby)

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Display the last name of all employees where the third letter of the name is a. (hints: like)

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

**Viva Questions:**

1. What’s the difference between WHERE, HAVING, and QUALIFY (in databases like Snowflake)? When should each be used?
2. If a column has NULL values, how do arithmetic operations behave in a SELECT query? How would you handle such cases?
3. Can we use arithmetic operations on string-type columns? Why or why not?
4. In what real-world situations would you use multiplication or division in a SELECT clause?
5. What is the importance of using aliases (AS keyword) when performing arithmetic in SELECT statements?

**Assignment Evaluation:**

0: Not Done [ ] 1: Incomplete [ ]

2: Late Complete [ ] 3: Needs Improvement [ ]

4: Complete [ ] 5: Well Done [ ]

**ACTIVITY SHEET 5: DATE:**

**Built-in-Functions**

**Objective**

After the completion of this exercise, the students will be able to do the following:

* Describe various types of built in functions available in SQL.
* Use character, number and date functions in SELECT statement.
* Describe the use of conversion functions.

**Built in Functions:**

Manipulate data items.

Accept arguments and return one value.

Act on each row returned.

Return one result per row.

May modify the data type.

Can be nested.

Accept arguments which can be a column or an expression

**Syntax**

Function\_name(arg1,…argn)

An argument can be one of the following

* User-supplied constant
* Variable value
* Column name

**SINGLE-ROW FUNCTIONS**

**CONVERSION**

**DATE**

**NUMBER**

**CHARACTER**

**GENERAL**

* Expression
* Character Functions: Accept character input and can return both character and number values.
* Number functions: Accept numeric input and return numeric values.
* Date Functions: Operate on values of the DATE data type.
* Conversion Functions: Convert a value from one type to another.

**Character Functions:**

Character Functions

Case-manipulation functions Character-manipulation functions

1. Lower 1. Concat
2. Upper 2. Substr
3. Initcap 3. Length

4. Instr

5. Lpad/Rpad

6. Trim

7. Repalce

|  |  |
| --- | --- |
| **Function** | **Purpose** |
| lower(column/expr) | Converts alpha character values to lowercase |
| upper(column/expr) | Converts alpha character values to uppercase |
| initcap(column/expr) | Converts alpha character values the to uppercase for the first letter of each word, all other letters in lowercase |
| concat(column1/expr1, column2/expr2) | Concatenates the first character to the second character |
| substr(column/expr,m,n) | Returns specified characters from character value starting at character position m, n characters long |
| length(column/expr) | Returns the number of characters in the expression |
| instr(column/expr,’string’,m,n) | Returns the numeric position of a named string |
| lpad(column/expr, n,’string’) | Pads the character value right-justified to a total width of n character positions |
| rpad(column/expr,’string’,m,n) | Pads the character value left-justified to a total width of n character positions |
| trim(leading/trailing/both, trim\_character FROM trim\_source) | Enables you to trim heading or string. trailing or both from a character |
| replace(text, search\_string, replacement\_string) |  |

**Example:**

lower(‘SQL Course’)🡪sql course

upper(‘SQL Course’)🡪SQL COURSE

initcap(‘SQL Course’)🡪Sql Course

SELECT ‘The job id for’|| upper(last\_name||’is’||lower(job\_id) AS “EMPLOYEE DETAILS” FROM employees;

SELECT employee\_id, last\_name, department\_id

FROM employees

WHERE LOWER(last\_name)=’higgins’;

|  |  |
| --- | --- |
| **Function** | **Result** |
| CONCAT(‘hello’, ‘world’) | helloworld |
| Substr(‘helloworld’,1,5) | Hello |
| Length(‘helloworld’) | 10 |
| Instr(‘helloworld’,’w’) | 6 |
| Lpad(salary,10,’\*’) | \*\*\*\*\*24000 |
| Rpad(salary,10,’\*’) | 24000\*\*\*\*\* |
| Trim(‘h’ FROM ‘helloworld’) | elloworld |

|  |  |  |
| --- | --- | --- |
| **Command** | **Query** | **Output** |
| initcap(char); | *select initcap(“hello”) from dual;* | Hello |
| lower (char);  upper (char); | *select lower (‘HELLO’) from dual;*  *select upper (‘hello’) from dual;* | Hello  HELLO |
| ltrim (char,[set]); | *select ltrim (‘cseit’, ‘cse’) from dual;* | IT |
| rtrim (char,[set]); | *select rtrim (‘cseit’, ‘it’) from dual;* | CSE |
| replace (char,search string, replace string); | *select replace (‘jack and jue’, ‘j’, ‘bl’) from dual;* | black and blue |
| substr (char,m,n); | *select substr (‘information’, 3, 4) from dual;* | form |

**Example:**

SELECT employee\_id, CONCAT (first\_name,last\_name) NAME , job\_id,LENGTH(last\_name), INSTR(last\_name,’a’) “contains’a’?”

FROM employees WHERE SUBSTR(job\_id,4)=’ERP’;

**NUMBER FUNCTIONS**

|  |  |
| --- | --- |
| **Function** | **Purpose** |
| round(column/expr, n) | Rounds the value to specified decimal |
| trunc(column/expr,n) | Truncates value to specified decimal |
| mod(m,n) | Returns remainder of division |

**Example**

|  |  |
| --- | --- |
| Function | Result |
| round(45.926,2) | 45.93 |
| trunc(45.926,2) | 45.92 |
| mod(1600,300) | 100 |

SELECT ROUND(45.923,2), ROUND(45.923,0), ROUND(45.923,-1) FROM dual;

**NOTE:**  Dual is a dummy table you can use to view results from functions and calculations.

SELECT TRUNC(45.923,2), TRUNC(45.923), TRUNC(45.923,-2) FROM dual;

SELECT last\_name,salary,MOD(salary,5000) FROM employees WHERE job\_id=’sa\_rep’;

**Working with Dates**

The Oracle database stores dates in an internal numeric format: century, year, month, day, hours, minutes, and seconds.

• The default date display format is DD-MON-RR.

– Enables you to store 21st-century dates in the 20th century by specifying only the last

two digits of the year

– Enables you to store 20th-century dates in the 21st century in the same way

**Example**

SELECT last\_name, hire\_date FROM employees WHERE hire\_date < '01-FEB-88;

**Working with Dates**

SYSDATE is a function that returns:

• Date

• Time

**Example**

**Display the current date using the DUAL table.**

SELECT SYSDATE FROM DUAL;

**Arithmetic with Dates**

• Add or subtract a number to or from a date for a resultant date value.

• Subtract two dates to find the number of days between those dates.

• Add hours to a date by dividing the number of hours by 24.

You can perform the following operations:

**Operation Result Description**

date + number Date Adds a number of days to a date

date – number Date Subtracts a number of days from a date

date – date Number of days Subtracts one date from another

date + number/24 Date Adds a number of hours to a date

**Example**

SELECT last\_name, (SYSDATE-hire\_date)/7 AS WEEKS

FROM employees

WHERE department\_id = 90;

**Date Functions**

Date functions operate on Oracle dates. All date functions return a value of DATE data type

except MONTHS\_BETWEEN, which returns a numeric value.

• **MONTHS\_BETWEEN(date1, date2)**::: Finds the number of months between date1 and date2. The result can be positive or negative. If date1 is later than date2, the result is positive; if date1 is earlier than date2, the result is negative. The noninteger part of the result represents a portion of the month.

• **ADD\_MONTHS(date, n)**::: Adds n number of calendar months to date. The value of n must be an integer and can be negative.

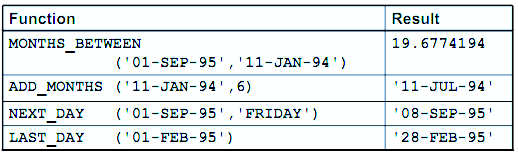
• **NEXT\_DAY(date, 'char')**::: Finds the date of the next specified day of the week ('char') following date. The value of char may be a number representing a day or a character string.

• **LAST\_DAY(date)**::: Finds the date of the last day of the month that contains date

• **ROUND(date[,'fmt'])**::: Returns date rounded to the unit that is specified by the format model fmt. If the format model fmt is omitted, date is rounded to the nearest day.

* **TRUNC(date[, 'fmt'])**::: Returns date with the time portion of the day truncated to the unit that is specified by the format model fmt. If the format model fmt is omitted, date is truncated to the nearest day.

**Using Date Functions**



**Example**

Display the employee number, hire date, number of months employed, sixmonth review date, first Friday after hire date, and last day of the hire month for all employees who have been employed for fewer than 70 months.

SELECT employee\_id, hire\_date,MONTHS\_BETWEEN (SYSDATE, hire\_date) TENURE,ADD\_MONTHS (hire\_date, 6) REVIEW,NEXT\_DAY (hire\_date, 'FRIDAY'), LAST\_DAY(hire\_date)

FROM employees

WHERE MONTHS\_BETWEEN (SYSDATE, hire\_date) < 70;

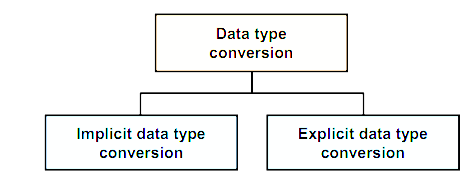
**Conversion Functions**

This covers the following topics:

• Writing a query that displays the current date

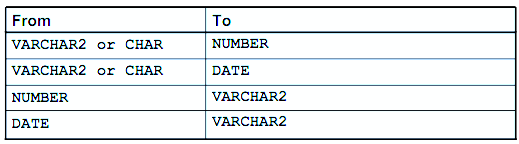
• Creating queries that require the use of numeric, character, and date functions

• Performing calculations of years and months of service for an employee

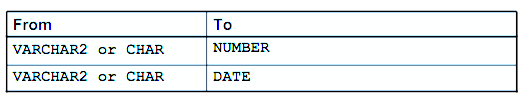


**Implicit Data Type Conversion**

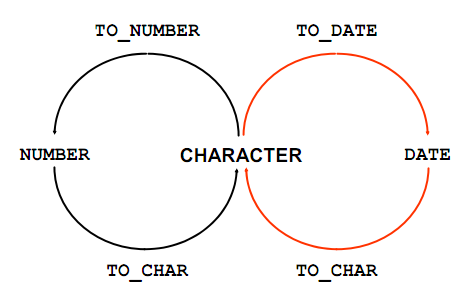
For assignments, the Oracle server can automatically convert the following:



For example, the expression hire\_date > '01-JAN-90' results in the implicit conversion from the string '01-JAN-90' to a date.

For expression evaluation, the Oracle Server can automatically convert the following:

**Explicit Data Type Conversion**



**SQL provides three functions to convert a value from one data type to another:**

**Example:**

**Using the TO\_CHAR Function with Dates**

TO\_CHAR(date, 'format\_model')

**The format model:**

• Must be enclosed by single quotation marks

• Is case-sensitive

• Can include any valid date format element

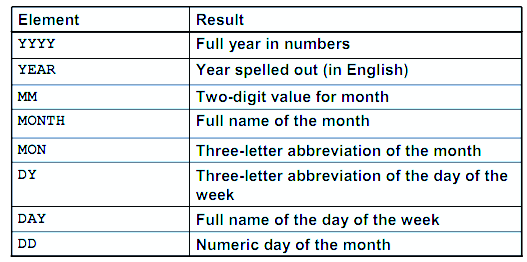
• Has an fm element to remove padded blanks or suppress leading zeros

• Is separated from the date value by a comma

SELECT employee\_id, TO\_CHAR(hire\_date, 'MM/YY') Month\_Hired

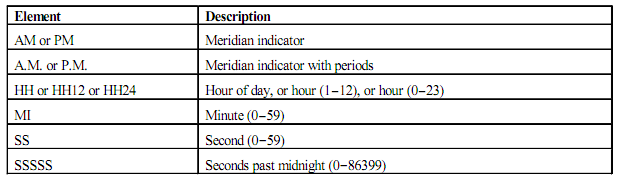
FROM employees WHERE last\_name = 'Higgins';

**Sample Format Date Elements**



**Date Format Elements:** Time Formats

Use the formats that are listed in the following tables to display time information and literals and to change numerals to spelled numbers.



**Example**

SELECT last\_name,

TO\_CHAR(hire\_date, 'fmDD Month YYYY') AS HIREDATE

FROM employees;

Modify example to display the dates in a format that appears as “Seventeenth of June 1987 12:00:00 AM.”

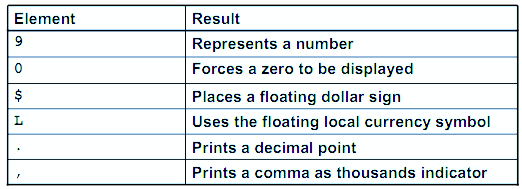
SELECT last\_name,

TO\_CHAR (hire\_date, 'fmDdspth "of" Month YYYY fmHH:MI:SS AM') HIREDATE

FROM employees;

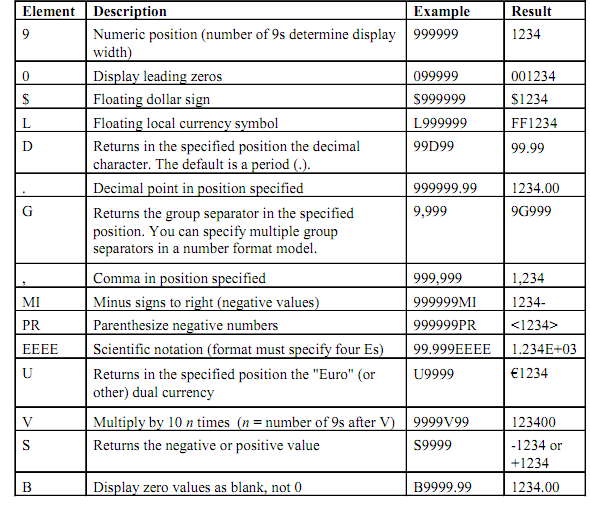
**Using the TO\_CHAR Function with Numbers**

TO\_CHAR(number, 'format\_model')  
These are some of the format elements that you can use with the TO\_CHAR function to display a number value as a character:



**Number Format Elements**

If you are converting a number to the character data type, you can use the following format elements:



SELECT TO\_CHAR(salary, '$99,999.00') SALARY

FROM employees

WHERE last\_name = 'Ernst';

**Using the TO\_NUMBER and TO\_DATE Functions**

• Convert a character string to a number format using the TO\_NUMBER function:

TO\_NUMBER(char[, 'format\_model']

• Convert a character string to a date format using the TO\_DATE function:

TO\_DATE(char[, 'format\_model']

• These functions have an fx modifier. This modifier specifies the exact matching for the character

argument and date format model of a TO\_DATE function.

The fx modifier specifies exact matching for the character argument and date format model of a TO\_DATE function:

• Punctuation and quoted text in the character argument must exactly match (except for case) the corresponding parts of the format model.

• The character argument cannot have extra blanks. Without fx, Oracle ignores extra blanks.

• Numeric data in the character argument must have the same number of digits as the corresponding element in the format model. Without fx, numbers in the character argument can omit leading zeros.

SELECT last\_name, hire\_date

FROM employees

WHERE hire\_date = TO\_DATE('May 24, 1999', 'fxMonth DD, YYYY');

**Self-Activity 5:**

**Consider the following table structure and solve the following SQL queries**

CREATE TABLE new\_employees (

emp\_id NUMBER,

first\_name VARCHAR2(20),

last\_name VARCHAR2(20),

job\_id VARCHAR2(10),

hire\_date DATE,

salary NUMBER(8,2)

);

1. Write a SQL query to extract the first 3 letters of last name.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. **Write a SQL query to display the full name of employees in lowercase.**

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a SQL query to round and truncate each salary to the nearest hundred.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a SQL query to display hire date as: “15th of September 2015 12:00:00 AM” format:

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a SQL query to Convert salary to a formatted character string with a dollar sign:

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a SQL query to show employees hired on “Tuesday”

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a SQL query to calculate the number of **full years** between hire date and today.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a SQL query to convert string to date and compare.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a SQL query to take a VARCHAR date in YYYYMMDD format and display it as “Weekday, Month DD, YYYY”:

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a SQL query to convert a numeric percentage (0.755) into a formatted string “75.5%”:

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

**Viva Question:**

1. How does MOD(m, n) behave with negative operands?
2. What does SYSDATE return, and how is it different from SYSTIMESTAMP (if you know)?
3. Differentiate between implicit and explicit data‑type conversion? Give examples.
4. How do TO\_CHAR, TO\_NUMBER, and TO\_DATE differ.
5. How would you format a number as currency (e.g., $12,345.67) in a single expression?

**Assignment Evaluation:**

**0: Not Done [ ] 1: Incomplete [ ]**

**2: Late Complete [ ] 3: Needs Improvement [ ]**

**4: Complete [ ] 5: Well Done [ ]**

**ACTIVITY SHEET 6: DATE:**

**USING SET OPERATORS**

**Objectives**

After the completion this exercise, the students should be able to do the following:

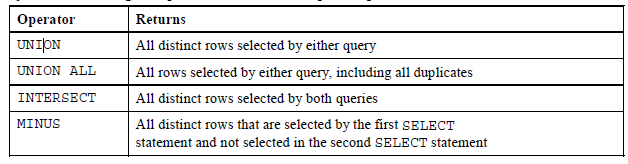
• Describe set operators

• Use a set operator to combine multiple queries into a single query

• Control the order of rows returned

The set operators combine the results of two or more component queries into one result.

Queries containing set operators are called *compound queries*.



**The tables used in this lesson are:**

• EMPLOYEES: Provides details regarding all current employees

• JOB\_HISTORY: Records the details of the start date and end date of the former job, and the job

identification number and department when an employee switch jobs

**UNION Operator**

**Guidelines**

• The number of columns and the data types of the columns being selected must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.

• UNION operates over all of the columns being selected.

• NULL values are not ignored during duplicate checking.

• The IN operator has a higher precedence than the UNION operator.

• By default, the output is sorted in ascending order of the first column of the SELECT clause.

**Example:**

Display the current and previous job details of all employees. Display each employee only once.

SELECT employee\_id, job\_id FROM employees UNION SELECT employee\_id, job\_id

FROM job\_history;

**Example:**

SELECT employee\_id, job\_id, department\_id

FROM employees

UNION

SELECT employee\_id, job\_id, department\_id

FROM job\_history;

**UNION ALL Operator**

**Guidelines**

The guidelines for UNION and UNION ALL are the same, with the following two exceptions that pertain to UNION ALL:

• Unlike UNION, duplicate rows are not eliminated and the output is not sorted by default.

• The DISTINCT keyword cannot be used.

**Example:**

Display the current and previous departments of all employees.

SELECT employee\_id, job\_id, department\_id

FROM employees

UNION ALL

SELECT employee\_id, job\_id, department\_id

FROM job\_history

ORDER BY employee\_id;

**INTERSECT Operator**

**Guidelines**

• The number of columns and the data types of the columns being selected by the SELECT statements in the queries must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.

• Reversing the order of the intersected tables does not alter the result.

• INTERSECT does not ignore NULL values.

**Example:**

Display the employee IDs and job IDs of those employees who currently have a job title that is the same as their job title when they were initially hired (that is, they changed jobs but have now gone back to

doing their original job).

SELECT employee\_id, job\_id FROM employees

INTERSECT

SELECT employee\_id, job\_id

FROM job\_history;

**Example**

SELECT employee\_id, job\_id, department\_id

FROM employees

INTERSECT

SELECT employee\_id, job\_id, department\_id

FROM job\_history;

**MINUS Operator**

**Guidelines**

• The number of columns and the data types of the columns being selected by the SELECT statements in the queries must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.

• All of the columns in the WHERE clause must be in the SELECT clause for the MINUS operator to work.

**Example:**

Display the employee IDs of those employees who have not changed their jobs even once.

SELECT employee\_id,job\_id

FROM employees

MINUS

SELECT employee\_id,job\_id

FROM job\_history;

**SELF ACTIVITY 6:**

1. Consider following tables and answer the question.

**STUDENT FACULTY**

|  |  |
| --- | --- |
| **FNAME1** | **LNAME1** |
| Aisha | Arora |
| Bikash | Dutta |
| Makku | Singh |
| Rajiv | chopra |

|  |  |
| --- | --- |
| **FNAME2** | **LNAME2** |
| Rajiv | chopra |
| Karan | Rao |
| Sanjay | Krishna |
| Mukesh | Singhal |

1. Remove the duplicate rows from the list. Apply the appropriate operator.

|  |
| --- |
| **Ans:** |
| **Query:** |
| **Output:** |

1. For the same table print the last name of student and faculty without removing the duplicate values.

|  |
| --- |
| **Ans:** |
| **Query:** |
| **Output:** |

1. For the above tables return the rows which exist in both student and faculty tables.

|  |
| --- |
| **Ans:** |
| **Query:** |
| **Output:** |

1. Write a SQL query to return the rows present in the student table but not in the faculty table.

|  |
| --- |
| **Ans:** |
| **Query:** |
| **Output:** |

**Viva Questions:**

1. List out the limitations and factors that can affect query performance, result accuracy, and overall usability.
2. How is UNION different from JOIN in SQL?
3. How do set operations impact performance compared to JOINs and subqueries?
4. What will happen if the number of columns doesn't match in a UNION query?
5. What happens if you use UNION on tables with large text (CLOB) or binary columns? What are the risks?
6. How do NULL values affect set operations like INTERSECT and MINUS? Are NULL = NULL in this context?

**Assignment Evaluation:**

**0: Not Done [ ] 1: Incomplete [ ]**

**2: Late Complete [ ] 3: Needs Improvement [ ]**

**4: Complete [ ] 5: Well Done [ ]**

**ACTIVITY SHEET 7: DATE:**

**SQL AGGREGATE FUNCTIONS**

**Objectives**

After the completion of this exercise, the students be will be able to do the following:

• Identify the available group functions

• Describe the use of group functions

• Group data by using the GROUP BY clause

• Include or exclude grouped rows by using the HAVING clause

**What Are Group Functions?**

Group functions operate on sets of rows to give one result per group

**Types of Group Functions:**

• AVG

• COUNT

• MAX

• MIN

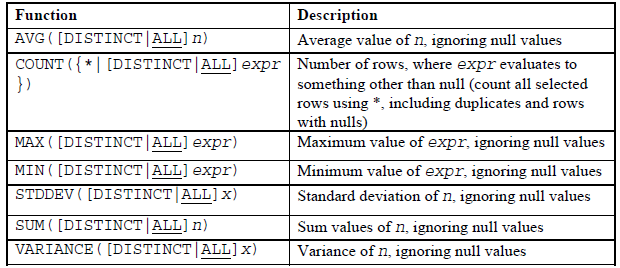
• STDDEV

• SUM

• VARIANCE

Each of the functions accepts an argument. The following table identifies the options that

you can use in the syntax:



**Group Functions: Syntax**

SELECT [*column*,] *group\_function(column), ...*

FROM *table*

[WHERE *condition*]

[GROUP BY *column*]

[ORDER BY *column*];

**Guidelines for Using Group Functions**

• DISTINCT makes the function consider only nonduplicate values; ALL makes it

consider every value, including duplicates. The default is ALL and therefore does not

need to be specified.

• The data types for the functions with an expr argument may be CHAR, VARCHAR2,

NUMBER, or DATE.

• All group functions ignore null values.

**Using the AVG and SUM Functions**

You can use AVG and SUM for numeric data.

SELECT AVG(salary), MAX(salary),

MIN(salary), SUM(salary)

FROM employees

WHERE job\_id LIKE '%REP%';

**Using the MIN and MAX Functions**

You can use MIN and MAX for numeric, character, and date data types.

SELECT MIN(hire\_date), MAX(hire\_date) FROM employees;

You can use the MAX and MIN functions for numeric, character, and date data types.

The following example displays the employee last name that is first and the employee last name that is last in an alphabetized list of all employees:

SELECT MIN(last\_name), MAX(last\_name) FROM employees;

**Note:**The AVG, SUM, VARIANCE, and STDDEV functions can be used only with numeric

data types. MAX and MIN cannot be used with LOB or LONG data types.

**Using the COUNT Function**

COUNT (\*) returns the number of rows in a table:

SELECT COUNT (\*)

FROM employees

WHERE department\_id = 50;

COUNT(*expr*) returns the number of rows with nonnull values for the *expr*:

SELECT COUNT(commission\_pct)

FROM employees

WHERE department\_id = 80;

**Using the DISTINCT Keyword**

• COUNT (DISTINCT expr) returns the number of distinct non-null values of the *expr*.

• To display the number of distinct department values in the EMPLOYEES table:

SELECT COUNT (DISTINCT department\_id) FROM employees;

Use the DISTINCT keyword to suppress the counting of any duplicate values in a column.

**Group Functions and Null Values**

Group functions ignore null values in the column:

SELECT AVG(commission\_pct) FROM employees;

The NVL function forces group functions to include null values:

SELECT AVG (NVL (commission\_pct, 0)) FROM employees;

**Creating Groups of Data**

To divide the table of information into smaller groups. This can be done by using the GROUP BY clause.

**GROUP BY Clause Syntax**

SELECT *column*, *group\_function(column)*

FROM *table*

[WHERE *condition*]

[GROUP BY *group\_by\_expression*]

[ORDER BY *column*];

**In the syntax:**

*group\_by\_expression* specifies columns whose values determine the basis for grouping rows

**Guidelines**

• If you include a group function in a SELECT clause, you cannot select individual results as well, *unless* the individual column appears in the GROUP BY clause. You receive an error message if you fail to include the column list in the GROUP BY clause.

• Using a WHERE clause, you can exclude rows before dividing them into groups.

• You must include the *columns* in the GROUP BY clause.

• You cannot use a column alias in the GROUP BY clause.

**Using the GROUP BY Clause**

All columns in the SELECT list that are not in group functions must be in the GROUP BY clause.

SELECT department\_id, AVG(salary)

FROM employees

GROUP BY department\_id ;

The GROUP BY column does not have to be in the SELECT list.

SELECT AVG(salary) FROM employees GROUP BY department\_id ;

You can use the group function in the ORDER BY clause:

SELECT department\_id, AVG(salary) FROM employees GROUP BY department\_id ORDER BY AVG(salary);

**Grouping by More Than One Column**

SELECT department\_id dept\_id, job\_id, SUM(salary) FROM employees

GROUP BY department\_id, job\_id ;

**Illegal Queries Using Group Functions**

Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP

**BY clause:**

SELECT department\_id, COUNT(last\_name) FROM employees;

You can correct the error by adding the GROUP BY clause:

SELECT department\_id, count(last\_name) FROM employees GROUP BY department\_id;

You cannot use the WHERE clause to restrict groups.

• You use the HAVING clause to restrict groups.

• You cannot use group functions in the WHERE clause.

SELECT department\_id, AVG(salary) FROM employees WHERE AVG(salary) > 8000

GROUP BY department\_id;

You can correct the error in the example by using the HAVING clause to restrict groups:

SELECT department\_id, AVG(salary) FROM employees

HAVING AVG(salary) > 8000 GROUP BY department\_id;

**Restricting Group Results**

With the HAVING Clause .When you use the HAVING clause, the Oracle server restricts groups as follows:

1. Rows are grouped.

2. The group function is applied.

3. Groups matching the HAVING clause are displayed.

**Using the HAVING Clause**

SELECT department\_id, MAX(salary) FROM employees

GROUP BY department\_idHAVING MAX(salary)>10000 ;

The following example displays the department numbers and average salaries for those departments with a maximum salary that is greater than $10,000:

SELECT department\_id, AVG(salary) FROM employees GROUP BY department\_id HAVING max(salary)>10000;

Example displays the job ID and total monthly salary for each job that has a total payroll exceeding $13,000. The example excludes sales representatives and sorts the list by the total monthly salary.

SELECT job\_id, SUM(salary) PAYROLL FROM employees WHERE job\_id NOT LIKE '%REP%'

GROUP BY job\_id HAVING SUM(salary) > 13000 ORDER BY SUM(salary);

**Nesting Group Functions**

**Display the maximum average salary:**

Group functions can be nested to a depth of two. The slide example displays the maximum average salary.

SELECT MAX(AVG(salary)) FROM employees GROUP BY department\_id;

**Complete Select Syntax:**

SELECT *column*, *group\_function* FROM *table* [WHERE *condition*][GROUP BY *group\_by\_expression*][HAVING *group\_condition*][ORDER BY *column*];

**SELF ACTIVITY 7:**

Consider following Employee Table and solve following queries.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| EMPLOYEE\_ID | FIRST\_NAME | LAST\_NAME | EMAIL | PHONE\_NUMBER | HIRE\_DATE | JOB\_ID | SALARY | COMMISSION\_PCT | MANAGER\_ID | DEPARTMENT\_ID |
| 100 | Steven | King | SKING | 515.123.4567 | 1987-06-17 | AD\_PRES | 24000.00 | 0.00 | 0 | 90 |
| 101 | Neena | Kochhar | NKOCHHAR | 515.123.4568 | 1987-06-18 | AD\_VP | 17000.00 | 0.00 | 100 | 90 |
| 102 | Lex | De Haan | LDEHAAN | 515.123.4569 | 1987-06-19 | AD\_VP | 17000.00 | 0.00 | 100 | 90 |
| 103 | Alexander | Hunold | AHUNOLD | 590.423.4567 | 1987-06-20 | IT\_PROG | 9000.00 | 0.00 | 102 | 60 |
| 104 | Bruce | Ernst | BERNST | 590.423.4568 | 1987-06-21 | IT\_PROG | 6000.00 | 0.00 | 103 | 60 |
| 105 | David | Austin | DAUSTIN | 590.423.4569 | 1987-06-22 | IT\_PROG | 4800.00 | 0.00 | 103 | 60 |
| 106 | Valli | Pataballa | VPATABAL | 590.423.4560 | 1987-06-23 | IT\_PROG | 4800.00 | 0.00 | 103 | 60 |
| 107 | Diana | Lorentz | DLORENTZ | 590.423.5567 | 1987-06-24 | IT\_PROG | 4200.00 | 0.00 | 103 | 60 |

1. Write a query to list the number of jobs available in the employees table.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a query to get the total salaries payable to employees.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a query to get the minimum salary from employees table.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a query to get the maximum salary of an employee working as a Programmer.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a query to get the average salary and number of employees working the department 90.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a query to get the highest, lowest, sum, and average salary of all employees.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a query to get the number of employees with the same job.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a query to get the difference between the highest and lowest salaries.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a query to find the manager ID and the salary of the lowest-paid employee for that manager.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Consider the following table CITY (ID NUMBER, NAME VARCHAR2 (17), COUNTRYCODE VARCHAR2 (3), DISTRICT VARCHAR2(20), POPULATION NUMBER);
2. Query a count of the number of cities in CITY having a Population larger than  100,000.

ii) For the same table City, Query the total population of all cities in **CITY** where District is **California**.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

**Viva Questions:**

**1.** Why is HAVING used with aggregate functions but not WHERE? Explain with an example.

2. Can we use aggregate functions on text columns like VARCHAR? Why or why not?

3. What would be the result of using an aggregate function without GROUP BY? When is this appropriate?

4. What’s the difference between COUNT (\*), COUNT (column), and COUNT (DISTINCT column)?

5. How does SUM() treat NULL values in the column being summed?

**Assignment Evaluation:**

**0: Not Done [ ] 1: Incomplete [ ]**

**2: Late Complete [ ] 3: Needs Improvement [ ]**

**4: Complete [ ] 5: Well Done [ ]**

**Activity Sheet 8: Date:**

**SQL Joins**

**Objective**

After the completion of this exercise, the students will be able to do the following:

• Write SELECT statements to access data from more than one table using equality and nonequality joins

• View data that generally does not meet a join condition by using outer joins

• Join a table to itself by using a self join

Sometimes you need to use data from more than one table.

**Cartesian Products**

• A Cartesian product is formed when:

– A join condition is omitted

– A join condition is invalid

– All rows in the first table are joined to all rows in the second table

• To avoid a Cartesian product, always include a valid join condition in a WHERE clause.

A Cartesian product tends to generate a large number of rows, and the result is rarely useful. You should always include a valid join condition in a WHERE clause, unless you have a specific need to combine all rows from all tables.

Cartesian products are useful for some tests when you need to generate a large number of rows to

simulate a reasonable amount of data.

**Example:**

To displays employee last name and department name from the EMPLOYEES and DEPARTMENTS tables.

SELECT last\_name, department\_name dept\_name

FROM employees, departments;

**Types of Joins**

• Equijoin

• Non-equijoin

• Outer join

• Self join

• Cross joins

• Natural joins

• Using clause

• Full or two sided outer joins

• Arbitrary join conditions for outer joins

**Joining Tables Using Oracle Syntax**

SELECT table1.column, table2.column

FROM table1, table2

WHERE table1.column1 = table2.column2;

Write the join condition in the WHERE clause.

• Prefix the column name with the table name when the same column name appears in more than one table.

**Guidelines**

• When writing a SELECT statement that joins tables, precede the column name with the table name for clarity and to enhance database access.

• If the same column name appears in more than one table, the column name must be prefixed with the table name.

• To join n tables together, you need a minimum of n-1 join conditions. For example, to join four tables, a minimum of three joins is required. This rule may not apply if your table has a concatenated primary key, in which case more than one column is required to uniquely identify each row

**What is an Equijoin?**

To determine an employee’s department name, you compare the value in the DEPARTMENT\_ID column in the EMPLOYEES table with the DEPARTMENT\_ID values in the DEPARTMENTS table.

The relationship between the EMPLOYEES and DEPARTMENTS tables is an equijoin—that is, values in the DEPARTMENT\_ID column on both tables must be equal. Frequently, this type of join involves primary and foreign key complements.

Note: Equijoins are also called simple joins or inner joins

SELECTemployees.employee\_id,employees.last\_name,employees.department\_id, departments.department\_id,departments.location\_id

FROM employees, departments

WHERE employees.department\_id = departments.department\_id;

**Additional Search Conditions**

**Using the AND Operator**

**Example:**

To display employee Matos’department number and department name, you need an additional condition in the WHERE clause.

SELECT last\_name, employees.department\_id,

department\_name

FROM employees, departments

WHERE employees.department\_id = departments.department\_id AND last\_name = ’Matos’;

**Qualifying Ambiguous Column Names**

• Use table prefixes to qualify column names that are in multiple tables.

• Improve performance by using table prefixes.

• Distinguish columns that have identical names but reside in different tables by using column aliases.

**Using Table Aliases**

• Simplify queries by using table aliases.

• Improve performance by using table prefixes

**Example:**

SELECT e.employee\_id, e.last\_name, e.department\_id,

d.department\_id, d.location\_id

FROM employees e , departments d

WHERE e.department\_id = d.department\_id;

**Joining More than Two Tables**

To join n tables together, you need a minimum of n-1 join conditions. For example, to join three

tables, a minimum of two joins is required.

**Example:**

To display the last name, the department name, and the city for each employee, you have to join the EMPLOYEES, DEPARTMENTS, and LOCATIONS tables.

SELECT e.last\_name, d.department\_name, l.city

FROM employees e, departments d, locations l

WHERE e.department\_id = d.department\_id

AND d.location\_id = l.location\_id;

**Non-Equijoins**

A non-equijoin is a join condition containing something other than an equality operator.The relationship between the EMPLOYEES table and the JOB\_GRADES table has an example of a non-equijoin. A relationship between the two tables is that the SALARY column in the EMPLOYEES table must be between the values in the LOWEST\_SALARY and HIGHEST\_SALARY columns of the JOB\_GRADES table. The relationship is obtained using an operator other than equals (=).

**Example:**

SELECT e.last\_name, e.salary, j.grade\_level

FROM employees e, job\_grades j

WHERE e.salary

BETWEEN j.lowest\_sal AND j.highest\_sal;

**Outer Joins**

**Syntax**

• You use an outer join to also see rows that do not meet the join condition.

• The Outer join operator is the plus sign (+).

SELECT table1.column, table2.column

FROM table1, table2

WHERE table1.column(+) = table2.column;

SELECT table1.column, table2.column

FROM table1, table2

WHERE table1.column = table2.column(+);

The missing rows can be returned if an outer join operator is used in the join condition. The operator is a plus sign enclosed in parentheses (+), and it is placed on the “side” of the join that is deficient in information. This operator has the effect of creating one or more null rows, to which one or more rows from the no deficient table can be joined.

**Example:**

SELECT e.last\_name, e.department\_id, d.department\_name

FROM employees e, departments d

WHERE e.department\_id(+) = d.department\_id ;

**Outer Join Restrictions**

• The outer join operator can appear on only one side of the expression—the side that has information missing. It returns those rows from one table that have no direct match in the other table.

• A condition involving an outer join cannot use the IN operator or be linked to another condition by the OR operator

**Self Join**

Sometimes you need to join a table to itself.

**Example:**

To find the name of each employee’s manager, you need to join the EMPLOYEES table to itself, or perform a self join.

SELECT worker.last\_name || ’ works for ’

|| manager.last\_name

FROM employees worker, employees manager

WHERE worker.manager\_id = manager.employee\_id ;

**Use a join to query data from more than one table.**

SELECT table1.column, table2.column

FROM table1

[CROSS JOIN table2] |

[NATURAL JOIN table2] |

[JOIN table2 USING (column\_name)] |

[JOIN table2

ON(table1.column\_name = table2.column\_name)] |

[LEFT|RIGHT|FULL OUTER JOIN table2

ON (table1.column\_name = table2.column\_name)];

**In the syntax:**

table1.column Denotes the table and column from which data is retrieved

CROSS JOIN Returns a Cartesian product from the two tables

NATURAL JOIN Joins two tables based on the same column name

JOIN table USING column\_name Performs an equijoin based on the column name

JOIN table ON table1.column\_name Performs an equijoin based on the condition in the ON clause = table2.column\_name

**LEFT/RIGHT/FULL OUTER**

**Creating Cross Joins**

• The CROSS JOIN clause produces the crossproduct of two tables.

• This is the same as a Cartesian product between the two tables.

**Example:**

SELECT last\_name, department\_name

FROM employees

CROSS JOIN departments ;

SELECT last\_name, department\_name

FROM employees, departments;

**Creating Natural Joins**

• The NATURAL JOIN clause is based on all columns in the two tables that have the same name.

• It selects rows from the two tables that have equal values in all matched columns.

• If the columns having the same names have different data types, an error is returned.

**Example:**

SELECT department\_id, department\_name,

location\_id, city

FROM departments

NATURAL JOIN locations ;

LOCATIONS table is joined to the DEPARTMENT table by the LOCATION\_ID column, which is the only column of the same name in both tables. If other common columns were present, the join would have used them all.

**Example:**

SELECT department\_id, department\_name,

location\_id, city

FROM departments

NATURAL JOIN locations

WHERE department\_id IN (20, 50);

**Creating Joins with the USING Clause**

• If several columns have the same names but the data types do not match, the NATURAL JOINclause can be modified with the USING clause to specify the columns that should be used for an equijoin.

• Use the USING clause to match only one column when more than one column matches.

• Do not use a table name or alias in the referenced columns.

• The NATURAL JOIN and USING clauses are mutually exclusive.

**Example:**

SELECT l.city, d.department\_name

FROM locations l JOIN departments d USING (location\_id)

WHERE location\_id = 1400;

**EXAMPLE:**

SELECT e.employee\_id, e.last\_name, d.location\_id

FROM employees e JOIN departments d

USING (department\_id) ;

**Creating Joins with the ON Clause**

• The join condition for the natural join is basically an equijoin of all columns with the same name.

• To specify arbitrary conditions or specify columns to join, the ON clause is used.

• The join condition is separated from other searchconditions.

• The ON clause makes code easy to understand.

**Example:**

SELECT e.employee\_id, e.last\_name, e.department\_id,

d.department\_id, d.location\_id

FROM employees e JOIN departments d

ON (e.department\_id = d.department\_id);

**EXAMPLE:**

SELECT e.last\_name emp, m.last\_name mgr

FROM employees e JOIN employees m

ON (e.manager\_id = m.employee\_id);

INNER Versus OUTER Joins

• A join between two tables that returns the results of the inner join as well as unmatched rows left (or right) tables is a left (or right) outer join.

• A join between two tables that returns the results of an inner join as well as the results of a left and right join is a full outer join.

**LEFT OUTER JOIN**

**Example:**

**SELECT e.last\_name, e.department\_id, d.department\_name**

FROM employees e

LEFT OUTER JOIN departments d

ON (e.department\_id = d.department\_id) ;

**Example of LEFT OUTER JOIN**

This query retrieves all rows in the EMPLOYEES table, which is the left table even if there is no match in the DEPARTMENTS table.

SELECT e.last\_name, e.department\_id, d.department\_name

FROM employees e, departments d

WHERE d.department\_id (+) = e.department\_id;

**RIGHT OUTER JOIN**

**Example:**

SELECT e.last\_name, e.department\_id, d.department\_name

FROM employees e

RIGHT OUTER JOIN departments d

ON (e.department\_id = d.department\_id) ;

This query retrieves all rows in the DEPARTMENTS table, which is the right table even if there is no match in the EMPLOYEES table.

This query was completed in earlier releases as follows:

SELECT e.last\_name, e.department\_id, d.department\_name

FROM employees e, departments d

WHERE d.department\_id = e.department\_id (+);

**FULL OUTER JOIN**

**Example:**

SELECT e.last\_name, e.department\_id, d.department\_name

FROM employees e

FULL OUTER JOIN departments d

ON (e.department\_id = d.department\_id) ;

This query retrieves all rows in the EMPLOYEES table, even if there is no match in the DEPARTMENTS table. It also retrieves all rows in the DEPARTMENTS table, even if there is no match in the EMPLOYEES table.

**Self-Activity 8:**

1. Consider following tables and answer the question.

**Tables:books**

|  |  |  |
| --- | --- | --- |
| **book\_id** | **title** | **author\_id** |
| 1 | The Hitchhiker's | 1 |
| 2 | Pride and Prejudice | 2 |
| 3 | The Lord of the Rings | 3 |

**Tables: authors**

|  |  |  |
| --- | --- | --- |
| **author\_id** | **name** | **nationality** |
| 1 | Douglas Adams | British |
| 2 | Jane Austen | British |
| 3 | J. R. R. Tolkien | British |

Write a query to retrieve a list of all books along with the author's name.

|  |
| --- |
| **Ans:** |
| **Query:** |
| **Output:** |

1. Consider following tables and answer the question.

**Tables: movies Tables: actors**

|  |  |  |
| --- | --- | --- |
| **movie\_id** | **title** | **release\_year** |
| 1 | The Godfather | 1972 |
| 2 | The Shawshank Redemption | 1994 |
| 3 | Pulp Fiction | 1994 |

|  |  |
| --- | --- |
| **actor\_id** | **name** |
| 1 | Marlon Brando |
| 2 | Al P acino |
| 3 | Tom Hanks |

Movie 3 (Pulp Fiction) has no actors listed in the actors table.

**Tables: cast** (movie\_id, actor\_id)

|  |  |
| --- | --- |
| **movie\_id** | **actor\_id** |
| 1 | 1 |
| 1 | 2 |
| 2 | 3 |
| 2 | 1 |

Write a query to retrieve a list of all movies, including the title, release year, and a comma-separated list of actors (if any) for each movie.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Consider following tables and answer the question.

**Tables:**

* **albums** (album\_id, title, artist\_id, genre\_id)

|  |  |  |  |
| --- | --- | --- | --- |
| **album\_id** | **title** | **artist\_id** | **genre\_id** |
| 1 | Dark Side of the Moon | 1 | 1 |
| 2 | Abbey Road | 2 | 2 |
| 3 | Thriller | 1 | 1 |
| 4 | Mock | 1 | 2 |

|  |  |
| --- | --- |
| **genre\_id** | **name** |
| 1 | Rock |
| 2 | Pop |
| 3 | R&B |
| 4 | Jazz |

* **artists** (artist\_id, name) **genres** (genre\_id, name)

|  |  |
| --- | --- |
| **artist\_id** | **name** |
| 1 | Pink Floyd |
| 2 | The Beatles |

Write a query to retrieve a list of all genres, including the

genre name and a comma-separated list of albums associated with that genre

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. **E-commerce Website (Multiple Joins)**

**Tables to create and insert data: ( atleast 5 rows for each table )**

* **products** (product\_id, name, price, category\_id)
* **categories** (category\_id, name)
* **orders** (order\_id, customer\_id, date)
* **order\_items** (order\_id, product\_id, quantity)

**Problem:** Write a query to retrieve a list of all orders, including the customer ID, order date, product details (name, price) for each item in the order, and the total order amount.

**Difficulty:** Medium (requires multiple joins and aggregations)

**Hint:** Utilize an INNER JOIN between orders and order\_items to connect orders with their corresponding items. Then, use another INNER JOIN between order\_items and products to retrieve product details. Finally, use aggregation functions like SUM to calculate the total order amount.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

**5. Library Management System (Advanced Filtering)**

**Tables to create and insert data: ( atleast 5 rows for each table )**

* **books** (book\_id, title, author\_id, publication\_year)
* **genres** (genre\_id, name)
* **book\_genres** (book\_id, genre\_id)

**Problem:** Write a query to retrieve a list of all books published after the year 2000, belonging to the "Science Fiction" genre.

**Difficulty:** Medium (requires filtering based on conditions and joining multiple tables)

**Hint:** Utilize an INNER JOIN between books and book\_genres to connect books with their associated genres. Then, use an INNER JOIN between book\_genres and genres to retrieve genre names. Finally, filter the results to include only books published after 2000 and belonging to the "Science Fiction" genre by using a WHERE clause with appropriate conditions.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

**6. Online Movie Database (Subqueries and Self-Joins)**

**Tables to create and insert data: ( atleast 5 rows for each table )**

* **movies** (movie\_id, title, release\_year, director\_id)
* **directors** (director\_id, name)
* **cast** (movie\_id, actor\_id)
* **actors** (actor\_id, name)

**Problem:** Write a query to retrieve a list of all movies directed by someone who also acted in at least one movie (excluding their own films).

**Difficulty:** Medium (introduces subqueries and potential self-joins)

**Hint:** You can approach this problem in two ways:

* **Subquery Method:** Use a subquery to identify director IDs of those who also appear as actors in any movie (excluding their own directed films). Then, use this subquery within the main query to filter movies based on the director ID.
* **Self-Join Method:** Utilize a self-join on the movies table, matching movies with their directors on the director\_id. In the second join condition, match movies with the cast table on movie\_id but exclude movies where the director\_id from the first join matches the actor\_id from the second join (to avoid including the director's own acting credits).

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

**Viva Questions:**

1. What are the benefits of using a subquery vs. a self-join when filtering a table?
2. What problem might occur if you forget to use a join condition in a multi-table query?
3. How does a JOIN between books and authors help avoid data duplication?
4. Compare JOIN … ON vs. JOIN … USING(...). When is each preferred?
5. How do NULL values in join columns affect INNER JOIN vs. OUTER JOIN results?

**Assignment Evaluation:**

**0: Not Done [ ] 1: Incomplete [ ]**

**2: Late Complete [ ] 3: Needs Improvement [ ]**

**4: Complete [ ] 5: Well Done [ ]**

**Activity Sheet 9: DATE:**

**SQL SUBQUERIES**

**Objectives**

After completing this lesson, you should be able to do the following:

• Define subqueries

• Describe the types of problems that subqueries can solve

• List the types of subqueries

• Write single-row and multiple-row subqueries

**Subquery Syntax**

SELECT *select\_list* FROM *table* WHERE *expr operator* (SELECT *select\_list* FROM *table*);

• The subquery (inner query) executes once before the main query (outer query).

• The result of the subquery is used by the main query.

A subquery is a SELECT statement that is embedded in a clause of another SELECT statement. You can build powerful statements out of simple ones by using subqueries. They can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself.

You can place the subquery in a number of SQL clauses, including the following:

• WHERE clause

• HAVING clause

• FROM clause

**In the syntax:**

*operator* includes a comparison condition such as >, =, or IN

**Note:** Comparison conditions fall into two classes: single-row operators

(>, =, >=, <, <>, <=) and multiple-row operators (IN, ANY, ALL). statement. The subquery generally executes first, and its output is used to complete the query condition for the main (or outer) query

**Using a Subquery**

SELECT last\_name FROM employees WHERE salary > (SELECT salary FROM employees

WHERE last\_name = 'Abel');

The inner query determines the salary of employee Abel. The outer query takes the result of the inner query and uses this result to display all the employees who earn more than this amount.

**Guidelines for Using Subqueries**

• Enclose subqueries in parentheses.

• Place subqueries on the right side of the comparison condition.

**•** The ORDER BY clause in the subquery is not needed unless you are performing Top-N analysis.

• Use single-row operators with single-row

Subqueries, and use multiple-row operators with multiple-row subqueries.

**Types of Subqueries:**

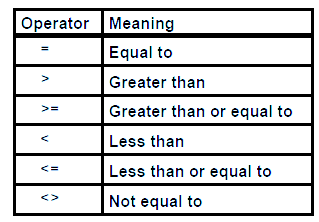
• Single-row subqueries: Queries that return only one row from the inner SELECT statement.

• Multiple-row subqueries: Queries that return more than one row from the inner SELECT statement.

**Single-Row Subqueries**

• Return only one row

• Use single-row comparison operators



**Example**

Display the employees whose job ID is the same as that of employee 141:

SELECT last\_name, job\_id FROM employees WHERE job\_id = (SELECT job\_id FROM employees

WHERE employee\_id = 141);

Displays employees whose job ID is the same as that of employee 141 and whose salary is greater

than that of employee 143.

SELECT last\_name, job\_id, salary FROM employeesWHERE job\_id =(SELECT job\_id FROM employees WHERE employee\_id = 141) AND salary > (SELECT salary FROM employees WHERE employee\_id = 143);

**Using Group Functions in a Subquery**

Displays the employee last name, job ID, and salary of all employees whose salary is equal to the minimum salary. The MIN group function returns a single value (2500) to the outer query.

SELECT last\_name, job\_id, salary FROM employees WHERE salary = (SELECT MIN(salary)

FROM employees);

**The HAVING Clause with Subqueries**

• The Oracle server executes subqueries first.

• The Oracle server returns results into the HAVING clause of the main query.

Displays all the departments that have a minimum salary greater than that of department 50.

SELECT department\_id, MIN(salary)

FROM employees

GROUP BY department\_id

HAVING MIN(salary) >

(SELECT MIN(salary)

FROM employees

WHERE department\_id = 50);

**Example**

**Find the job with the lowest average salary.**

SELECT job\_id, AVG(salary)

FROM employees

GROUP BY job\_id

HAVING AVG(salary) = (SELECT MIN(AVG(salary))

FROM employees

GROUP BY job\_id);

**What Is Wrong in this Statements?**

SELECT employee\_id, last\_name

FROM employees

WHERE salary =(SELECT MIN(salary) FROM employees GROUP BY department\_id);

Will This Statement Return Rows?

SELECT last\_name, job\_id

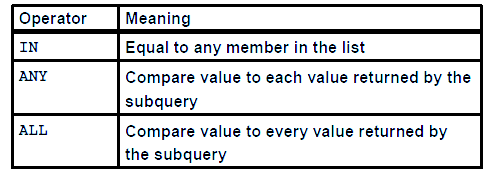
FROM employees

WHERE job\_id =(SELECT job\_id FROM employees WHERE last\_name = 'Haas');

**Multiple-Row Subqueries**

• Return more than one row

• Use multiple-row comparison operators



**Example**

Find the employees who earn the same salary as the minimum salary for each department.

SELECT last\_name, salary, department\_id FROM employees WHERE salary IN (SELECT MIN(salary)

FROM employees GROUP BY department\_id);

Using the ANY Operator in Multiple-Row Subqueries

SELECT employee\_id, last\_name, job\_id, salary FROM employees WHERE salary < ANY

(SELECT salary FROM employees WHERE job\_id = 'IT\_PROG') AND job\_id <> 'IT\_PROG';

Displays employees who are not IT programmers and whose salary is less than that of any IT programmer. The maximum salary that a programmer earns is $9,000.

< ANY means less than the maximum. >ANY means more than the minimum. =ANY is equivalent to IN.

**Using the ALL Operator in Multiple-Row Subqueries**

SELECT employee\_id, last\_name, job\_id, salary

FROM employees

WHERE salary < ALL (SELECT salary FROM employees WHERE job\_id = 'IT\_PROG')

AND job\_id <> 'IT\_PROG';

Displays employees whose salary is less than the salary of all employees with a job ID of IT\_PROG and whose job is not IT\_PROG.

ALL means more than the maximum, and <ALL means less than the minimum.

The NOT operator can be used with IN, ANY, and ALL operators.

**Null Values in a Subquery**

SELECT emp.last\_name FROM employees emp

WHERE emp.employee\_id NOT IN (SELECT mgr.manager\_id FROM employees mgr);

Notice that the null value as part of the results set of a subquery is not a problem if you use the IN operator. The IN operator is equivalent to =ANY. For example, to display the employees who have subordinates, use the following SQL statement:

SELECT emp.last\_name

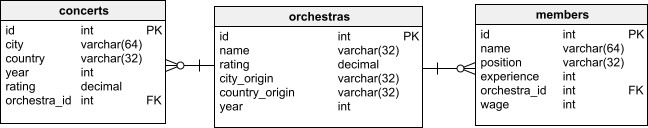
FROM employees emp

WHERE emp.employee\_id IN (SELECT mgr.manager\_id FROM employees mgr);

Display all employees who do not have any subordinates:

SELECT last\_name FROM employees

WHERE employee\_id NOT IN (SELECT manager\_id FROM employees WHERE manager\_id IS NOT NULL)

**Self-Activity 9:**

The following exercises use the orchestras dataset that contains three tables.

* The orchestras table stores all orchestras. The columns are id, name, rating, city\_origin, country\_origin, and year in which the orchestra was founded.
* The concerts table contains all concerts played by the orchestras. The columns are id, city, country, year, rating, and orchestra\_id (references the orchestras table).
* The members table stores the members of (i.e. musicians playing in) each orchestra. The columns are id, name, position (i.e. the instrument played), wage, experience, and orchestra\_id (references the orchestras table).

**Exercise 1: Select Orchestras with a City of Origin Where a Concert Was Held in 2013** Select the names of all orchestras that have the same city of origin as any city in which any orchestra performed in 2013.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

**Exercise 2: Select Members that Belong to High-Rated Orchestras**

Select the names and positions (i.e. instrument played) of all orchestra members that have above 10 years of experience and do not belong to orchestras with a rating below 8.0.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

**Exercise 3: Select Members Who Earn More Than Violinists**

Show the name and position of orchestra members who earn more than the average wage of all violinists.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

**Exercise 4: Select High-Rated Orchestras Newer Than the Chamber Orchestra**

Show the names of orchestras that were created after the 'Chamber Orchestra' and have a rating greater than 7.5.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

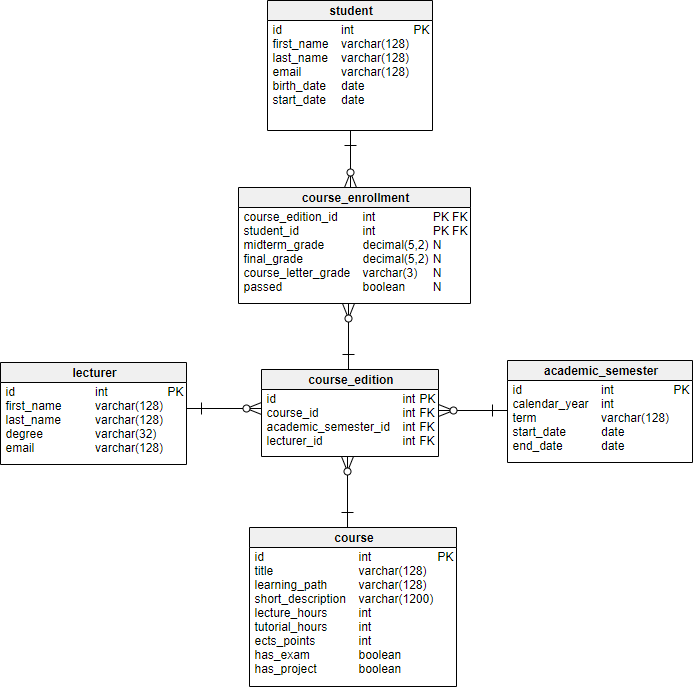
**Exercise 5: Select Players in Large Orchestras**

Show the name and number of members for each orchestra that has more members than the average membership of all orchestras in the table.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. **Dataset: University**

The following exercises use the university dataset, which contains six tables.



* The course table stores information about courses. The columns are id, title, learning\_path, short\_description, lecture\_hours, tutorial\_hours, ects\_points, has\_exam, and has\_project.
* The lecturer table stores information about lecturers. The columns are id, first\_name, last\_name, degree, and email.
* The student table contains information about students. The columns are id, first\_name, last\_name, email, birth\_date, and start\_date.
* The academic\_semester table contains information about each study semester. The columns are id, calendar\_year, term, start\_date, and end\_date.
* The course\_edition table contains information on which lecturers teach each course in each semester. The columns are id, course\_id (references the course table), academic\_semester\_id (references the academic\_semester table), and lecturer\_id (references the lecturer table).
* The course\_enrollment table contains information about students enrolled in each course. The columns are course\_edition\_id (references the course\_edition table), student\_id (references the student table), midterm\_grade, final\_grade, course\_letter\_grade, and passed.

**Exercise 6: Select Spring Term Courses**

Display the IDs and titles of all courses that took place during any spring term.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

**Exercise 7: Select All Students Who Passed At Least One Course**

Select the IDs and names of students who passed at least one course.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

**Exercise 8: Select the Lecturer(s) Teaching the Fewest Courses**

Find the lecturer(s) with the least number of courses taught. Display the lecturer’s first and last name and the number of courses they teach (as no\_of\_courses).

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

**Exercise 9: Select Students Enrolled in the Most Courses**

Find the student(s) enrolled in the greatest number of course editions. Display the student’s ID, first and last names, and the number of course editions they’ve been enrolled in (as no\_of\_course\_ed).

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

**Viva Questions**

1. In what scenarios does a subquery offer a clearer or more efficient solution than a JOIN?
2. How do correlated subqueries impact performance compared to uncorrelated subqueries?
3. Can subqueries be used to enforce business rules or data constraints? How effective is this approach compared to triggers or constraints?
4. How does the placement of a subquery (in SELECT, FROM, or WHERE clause) affect the query logic and performance?
5. Can a subquery return more than one row? What happens if it does in different contexts (e.g., scalar subquery in SELECT)?

**Assignment Evaluation:**

**0: Not Done [ ] 1: Incomplete [ ]**

**2: Late Complete [ ] 3: Needs Improvement [ ]**

**4: Complete [ ] 5: Well Done [ ]**

**Activity Sheet 10: Date:**

**PL/SQL**

While the SQL is the natural language of the DBA, it does not have any procedural capabilities such as looping and branching. For all this, oracle provides PL/SQL, it adds power to SQL and provides the user with all the facilities of a programming environment. It bridges the gap between database technology and procedural programming languages.

Execute the following command first

SQL>set serveroutput on;

PL/SQL Block consists of three sections:

 The Declaration section (optional).

 The Execution section (mandatory).

 The Exception (or Error) Handling section (optional).

Declaration Section:

The Declaration section of a PL/SQL Block starts with the reserved keyword DECLARE. This section is optional and is used to declare any placeholders like variables, constants, records and cursors, which are used to manipulate data in the execution section. Placeholders may be any of Variables, Constants and Records, which stores data temporarily. Cursors are also declared in this section.

Execution Section:

The Execution section of a PL/SQL Block starts with the reserved keyword BEGIN and ends with END. This is a mandatory section and is the section where the program logic is written to perform any task. The programmatic constructs like loops, conditional statement and SQL statements form the part of execution section.

Exception Section:

The Exception section of a PL/SQL Block starts with the reserved keyword EXCEPTION. This section is optional. Any errors in the program can be handled in this section, so that the PL/SQL Blocks terminates gracefully. If the PL/SQL Block contains exceptions that cannot be handled, the Block terminates abruptly with errors.

Every statement in the above three sections must end with a semicolon; PL/SQL blocks can be nested within other PL/SQL blocks. Comments can be used to document code.

**Sample PL/SQL Block Looks**

DECLARE

BEGIN

EXCEPTION

END;

In addition to SQL commands, PL/SQL can also process data usin flow of statements.the flow of control statements are classified into the following categories.

* Conditional control -Branching
* Iterative control - looping
* Sequential control

**BRANCHING in PL/SQL:**

Sequence of statements can be executed on satisfying certain condition.

If statements are being used and different forms of if are:

1.Simple IF

2.ELSIF

3.ELSE IF

**SIMPLE IF:**

**Syntax:**

IF condition THEN

statement1;

statement2;

END IF;

**IF-THEN-ELSE STATEMENT:**

**Syntax:**

IF condition THEN

statement1;

ELSE

statement2;

END IF;

**ELSIF STATEMENTS:**

**Syntax:**

IF condition1 THEN

statement1;

ELSIF condition2 THEN

statement2;

ELSIF condition3 THEN

statement3;

ELSE

statementn;

END IF;

**NESTED IF :**

**Syntax:**

IF condition THEN

statement1;

ELSE

IF condition THEN

statement2;

ELSE

statement3;

END IF;

END IF;

ELSE

statement3;

END IF;

**SELECTION IN PL/SQL(Sequential Controls)**

**SIMPLE CASE**

**Syntax:**

CASE SELECTOR

WHEN Expr1 THEN statement1;

WHEN Expr2 THEN statement2;

ELSE

Statement n;

END CASE;

**SEARCHED CASE:**

CASE

WHEN searchcondition1 THEN statement1;

WHEN searchcondition2 THEN statement2;

:

:

ELSE

statementn;

END CASE;

**ITERATIONS IN PL/SQL**

Sequence of statements can be executed any number of times using loop construct.

It is broadly classified into:

* Simple Loop
* For Loop
* While Loop

***SIMPLE LOOP***

**Syntax:**

LOOP

statement1;

EXIT [ WHEN Condition];

END LOOP;

**WHILE LOOP**

**Syntax:**

WHILE condition LOOP

statement1;

statement2;

END LOOP;

**FOR LOOP**

**Syntax:**

FOR counter IN [REVERSE]

LowerBound..UpperBound

LOOP

statement1;

statement2;

END LOOP;

**Example 1: (Simple Example)**

DECLARE

A varchar2(20);

BEGIN

select dept\_name into A from department where budget = 80000; dbms\_output.put\_line(A);

END;

**Example 2: (To use if...then…else...endif)**

DECLARE

B number(12,2);

BEGIN

select budget into B from department where dept\_name = 'Music'; if B > 5000 then

dbms\_output.put\_line('Good');

else

dbms\_output.put\_line('bad'); end if;

END;

**Example 3: (To use while loop)**

Declare

name varchar2(20):= ‘CSE’;

counter number(2):=5;

BEGIN

select dept\_name into name from department where budget=80000;

while counter>0

loop

dbms\_output.put\_line(name);

counter:=counter-1;

end loop;

END;

**Example 4: (To use while loop)**

/\* counter variable need not be declared Declare

name varchar2(20);

BEGIN

select dept\_name into name from department where budget=80000;

for counter in 1..5

loop

dbms\_output.put\_line(counter||'.'||name);

end loop;

END;

**We can also use for counter in reverse 1..5**

**Example 5: (To use goto statement)**

DECLARE

B number(12,2);

BEGIN

select budget into B from department where dept\_name = 'Music'; if B > 79000 then

goto good;

else

goto bad; end if;

<<good>> dbms\_output.put\_line('Good');

<<bad>> dbms\_output.put\_line('Bad');

END;

**PROCEDURES and Functions**

A procedure or function is a logically grouped set of SQL and PL/SQL statements that perform a specific task. They are essentially sub-programs. Procedures and functions are made up of,

* Declarative part
* Executable part
* Optional exception handling part

These procedures and functions do not show the errors.

**PROCEDURES – SYNTAX**

 create or replace procedure <procedure name> (argument {in,out,inout} datatype ) {is,as}

 variable declaration;

 constant declaration;

 begin

 PL/SQL subprogram body;

 exception

 exception PL/SQL block;

 end;

Example:

SQL> create procedure itsum(identity number, total number) is price number;

  2  null\_price exception;

  3  begin

  4  select actualprice into price from ititems where itemid=identity;

  5  if price is null then

  6  raise null\_price;

  7  else

  8  update ititems set actualprice=actualprice+total where itemid=identity;

  9  end if;

 10  exception

 11  when null\_price then

 12  dbms\_output.put\_line('price is null');

 13  end;

 14  /

Procedure created.

SQL> exec itsum(101, 500);

PL/SQL procedure successfully completed.

SQL> select \* from ititems;

 ITEMID   ACTUALPRICE      ORDID     PRODID

---------            -----------              ---------      ---------

      101             2500                      500          201

      102             3000                    1600          202

      103            4000                       600          202

**FUNCTIONS – SYNTAX**

 create or replace function <function name> (argument in datatype,……) return datatype {is,as}

 variable declaration;

 constant declaration;

 begin

 PL/SQL subprogram body;

 exception

 exception PL/SQL block;

 end;

Example:

SQL> create function aaa (trainnumber number) return number is

  2  trainfunction ittrain.tfare % type;

  3  begin

  4  select tfare into trainfunction from ittrain where tno=trainnumber;

  5  return(trainfunction);

  6  end;

  7  /

Function created.

SQL> set serveroutput on;

SQL> declare

  2  total number;

  3  begin

  4  total:=aaa (1001);

  5  dbms\_output.put\_line('Train fare is Rs. '||total);

  6  end;

  7  /

Train fare is Rs.550

PL/SQL procedure successfully completed.

**Self-Activity 10:**

1. Write a PL/SQL program to find the sum and product of two numbers.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a PL/SQL procedure to find the factorial of a nuber.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Find the department name from department table using pl/sql block

**Department Table:**

|  |  |  |
| --- | --- | --- |
| Deptid | dname | location |
| 101 | cse | main block |
| 102 | it | I block |

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Write a PL/SQL function to check whether a given number is Armstrong number or not.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

Viva Questions:

1. How do exceptions in PL/SQL contribute to robust programming?
2. How does PL/SQL handle context switch between SQL and procedural logic? Why does it matter?
3. How can recursion be handled in PL/SQL? Should it be used, or avoided? Why?
4. How can you log errors in PL/SQL without cluttering business logic?
5. What is an anonymous block in PL/SQL?

**Assignment Evaluation:**

**0: Not Done [ ] 1: Incomplete [ ]**

**2: Late Complete [ ] 3: Needs Improvement [ ]**

**4: Complete [ ] 5: Well Done [ ]**

**Activity Sheet 11: Date:**

**CREATING VIEWS, SYNONYMS, SEQUENCE, INDEX**

After the completion of this exercise, students will be able to do the following:

* Describe a view
* Create, alter the definition of, and drop a view
* Retrieve data through a view
* Insert, update, and delete data through a view
* Create and use an inline view

**View**

A view is a logical table based on a table or another view. A view contains no data but is like a window through which data from tables can be viewed or changed. The tables on which a view is based are called base tables.

**Advantages of Views**

* To restrict data access
* To make complex queries easy
* To provide data independence
* To present different views of the same data

**SYNTAX:**

The following are the SQL Objects and their syntax.

VIEWS:

CREATE VIEW <VIEWNAME> AS SELECT \* FROM <TABLENAME> WHERE

<CONDITION>;

Example:

Student table:

REGNO NAME MARK1 MARK2 TOTAL

130 ajay 90 90 180

126 aldo 95 96 191

76 guru 90 95 185

SQL> create view studentdetail as select regno, name, mark1 from student;

SYNONYMS:

A **synonym** is an alias for another database object, such as a table, view, sequence, or another synonym.

CREATE SYNONYM <SYNONYMS NAME> FOR <TABLENAME>;

CREATE TABLE product (product\_name VARCHAR2(25) PRIMARY KEY, product\_priceNUMBER(4,2), quantity\_on\_hand NUMBER(5,0),

last\_stock\_date DATE);.

SQL> CREATE SYNONYM prod FOR product;

SQL> SELECT \* FROM prod;

SQL> drop SYNONYM prod; Synonym dropped.

SQL> drop table product; Table dropped.

SEQUENCE:

A **sequence** generates unique numeric values, often used for primary key values.

CREATE SEQUENCE<SEQUENCENAME> START WITH <VALUE> INCREMENT BY <VALUE>;

SEQUENCE

Create a sequence and design the student table with the given attributes.

SQL> create table student(student\_id number, name varchar2(10),result varchar2(10));

Sequence Creation:

SQL> create sequence student\_seq start with 100 increment by 1; Sequence created.

SQL> insert into student values(student\_seq.nextval,'raja','pass'); 1 row created.

SQL> insert into student values(student\_seq.nextval,'ravi','pass'); 1 row created.

SQL> select \* from student; STUDENT\_ID NAME

100 raja pass

101 ravi pass

INDEXES:

An **index** improves the speed of data retrieval on one or more columns.

CREATE INDEX < INDEX\_NAME> ON TABLENAME (COLUMNNAME [, COLUMNNAME...]);

To create index on the Last Name column of the Employee table.

SQL> create table Employee(ID VARCHAR2(4 BYTE) NOT NULL, first\_name VARCHAR2(10 BYTE), Last\_Name VARCHAR2(10 BYTE), Start\_Date DATE,

End\_Date DATE, salary Number(8,2)); Table created.

SQL> CREATE INDEX LastNameIndex ON Employee (Last\_Name); Index Created.

SQL> drop index LastNameIndex; Index Dropped.

**Self-Activity 11:**

1. Create a view student\_total showing regno, name, and total from the student table.

Alter the view studentdetail to include mark2. Drop the studentdetail view.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Create a table employee\_details. Create a synonym emp for employee\_details.

Insert and select data using emp.Drop the synonym.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Create a sequence emp\_seq starting at 500 and incrementing by 5. Create a table employee(id NUMBER, name VARCHAR2(20)) . Insert 3 rows using emp\_seq.NEXTVAL**.** Display all data from the table.

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

1. Create a table book(book\_id NUMBER, title VARCHAR2(50), author VARCHAR2(30)). Create an index author\_idx on the author column. Drop the index author\_idx

|  |
| --- |
| **Ans:** |
| **Query :** |
| **Output:** |

Viva Questions:

1. How can views be used to implement row-level security in a multi-user database?
2. What are the potential risks or downsides of using updatable views?
3. In what scenarios might synonyms lead to confusion or maintenance issues in a large-scale application?
4. In what cases can an index make query performance worse instead of better?
5. Why might you choose a sequence over an auto-increment column in some databases, and vice versa?

**Assignment Evaluation:**

0: Not Done [ ] 1: Incomplete [ ]

2: Late Complete [ ] 3: Needs Improvement [ ]

4: Complete [ ] 5: Well Done [ ]

**Exercise 12: Working with advanced databases (Basic Operations) MongoDB**

* MongoDB is a NoSQL database. There are different types of NoSQL databases, so to be specific MongoDB is an open source document based NoSQL database.
* NoSQL databases are different than relational databases like MQSql. In relational database we need to create the table, define schema, set the data types of fields etc before you can actually insert the data. In NoSQL we can directly insert, delete and update.

**Install MongoDB on Windows**

* Step 1: Go to https://www.mongodb.com/download-center#atlas download as shown in the screenshot. A .msi file like this **mongodb-win32-x86\_64-2008plus-ssl-3.4.7-signed** will be downloaded in your system. Double click on the file to run the installer.
* Step 2: Click Next when the MongoDB installation windows pops up.
* Step 3: Accept the MongoDB user Agreement and click Next.
* Step 4: When the setup asks you to choose the Setup type, choose Complete.
* Step 5: Click Install to begin the installation.
* Step 6: Click Finish once the MongoDB installation is complete.

**MongoDB Configuration**

* **Step 1**: Locate the folder where you have installed MongoDB. If you have followed the above steps then you can find the folder at this location:
* C:\Program Files\MongoDB
* Here you need to create couple of folders that we need for MongoDB configuration.  
  1. Create two folders here, name them **data** and **log**.  
  2. Create another folder inside **data** and name it as **db**, that’s where all the data will be stored.  
  That’s it close the window.
* **Step 2**: Open command prompt (right click and run as administrator). Navigate to the **bin** folder of MongoDB as shown in the screenshot. The path to the bin folder may be different in your case based on where you have installed the MongoDB.
* Step3 : open command prompt type mongodb
* Step4 : open another command prompt and type mongo .

> show dbs

admin 0.000GB

local 0.000GB

We are creating a database “beginnersbook” so the command should be:

>use beginnersbook

creating a collection **user** and inserting a document in it.

> db.user.insert({name: "Chaitanya", age: 30})

WriteResult({ "nInserted" : 1 })

> show dbs

admin 0.000GB

beginnersbook 0.000GB

local 0.000GB

* **Creating the Collection in MongoDB**

**db.collection\_name.insert({key:value, key:value…})**

> use beginnersbookdb

switched to db beginnersbookdb

db.beginnersbook.insert({

name: "Chaitanya",

age: 30,

website: "beginnersbook.com"

})

To check whether the document is successfully inserted, type the following command. It shows all the documents in the given collection.  
Syntax: **db.collection\_name.find()**

> db.beginnersbook.find()

{ "\_id" : ObjectId("59bcb8c2415346bdc68a0a66"), "name" : "Chaitanya",

"age" : 30, "website" : "beginnersbook.com" }

This command shows the list of all the collections in the currently selected database.

> show collections

beginnersbook

To drop a collection , first connect to the database in which you want to delete collection and then type the following command to delete the collection:

db.collection\_name.drop()

For example I want to delete a collection names “teachers” in my database “beginnersbook.com”. To do this I would write the following commands in the given sequence.

> use beginnersbookdb

switched to db beginnersbookdb

> show collections

beginnersbook

students

teachers

> db.teachers.drop()

true

> show collections

beginnersbook

students

* **Updating Document using update () method**

**Syntax:**

db.collection\_name.update(criteria, update\_data)

**Example:**  
For example: Lets say I have a collection named “got” in the database “beginnersbookdb”. The documents inside “got” are:

> db.got.find().pretty()

{

"\_id" : ObjectId("59bd2e73ce524b733f14dd65"),

"name" : "Jon Snow",

"age" : 32

}

{

"\_id" : ObjectId("59bd2e8bce524b733f14dd66"),

"name" : "Khal Drogo",

"age" : 36

}

{

"\_id" : ObjectId("59bd2e9fce524b733f14dd67"),

"name" : "Sansa Stark",

"age" : 20

}

{

"\_id" : ObjectId("59bd2ec5ce524b733f14dd68"),

"name" : "Lord Varys",

"age" : 42

}

Now suppose if I want to update the name of Jon Snow with the name “Kit Harington”. The command for this would be:

db.got.update({"name":"Jon Snow"},{$set:{"name":"Kit Harington"}})

* **Syntax of remove() method:**

db.collection\_name.remove(delete\_criteria)

**Delete Document using remove() method**

Lets say I have a collection students in my MongoDB database named beginnersbookdb. The documents in students collection are:

> db.students.find().pretty()

{

"\_id" : ObjectId("59bcecc7668dcce02aaa6fed"),

"StudentId" : 1001,

"StudentName" : "Steve",

"age" : 30

}

{

"\_id" : ObjectId("59bcecc7668dcce02aaa6fef"),