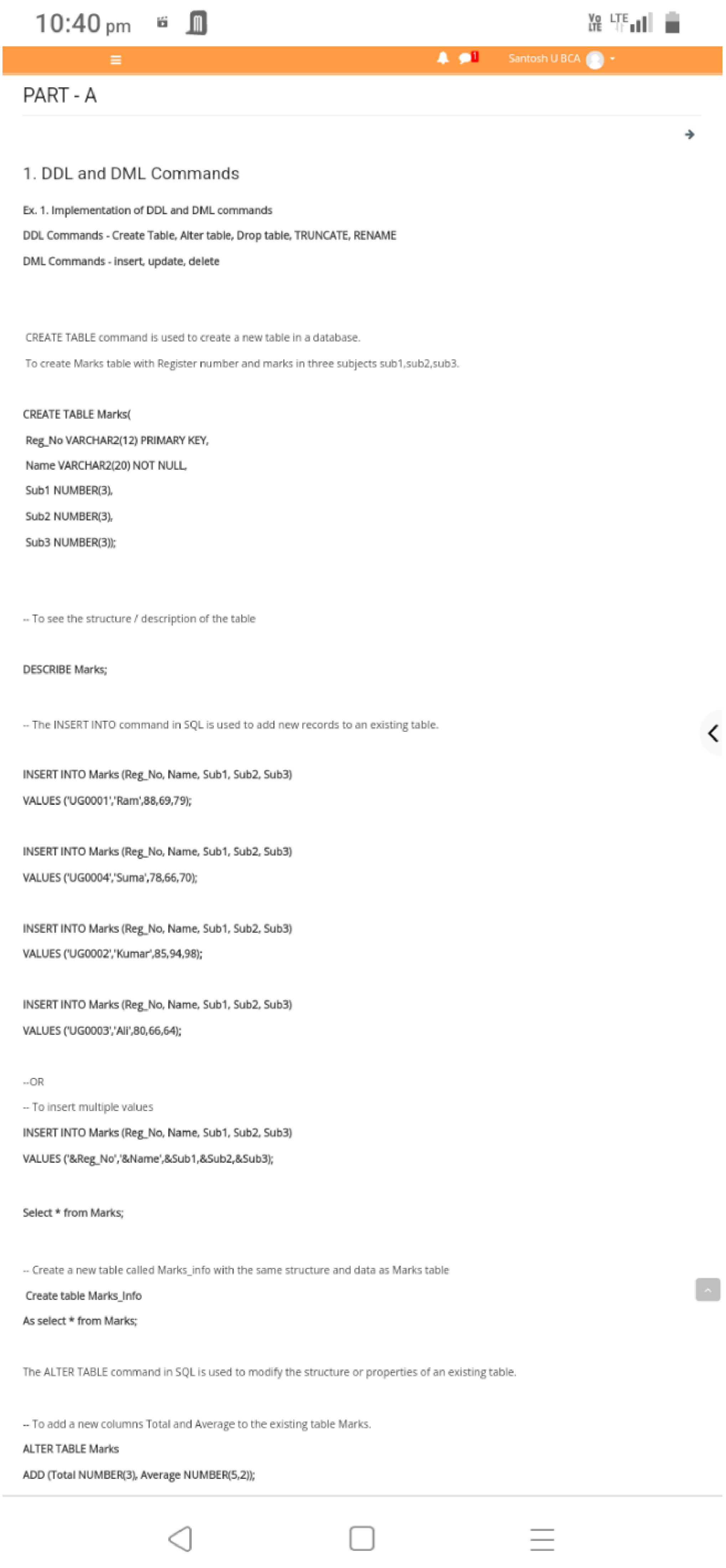


PART - A

1. DDL and DML Commands

Ex. 1. Implementation of DDL and DML commands

DDL Commands - Create Table, Alter table, Drop table, TRUNCATE, RENAME

DML Commands - insert, update, delete

CREATE TABLE command is used to create a new table in a database.

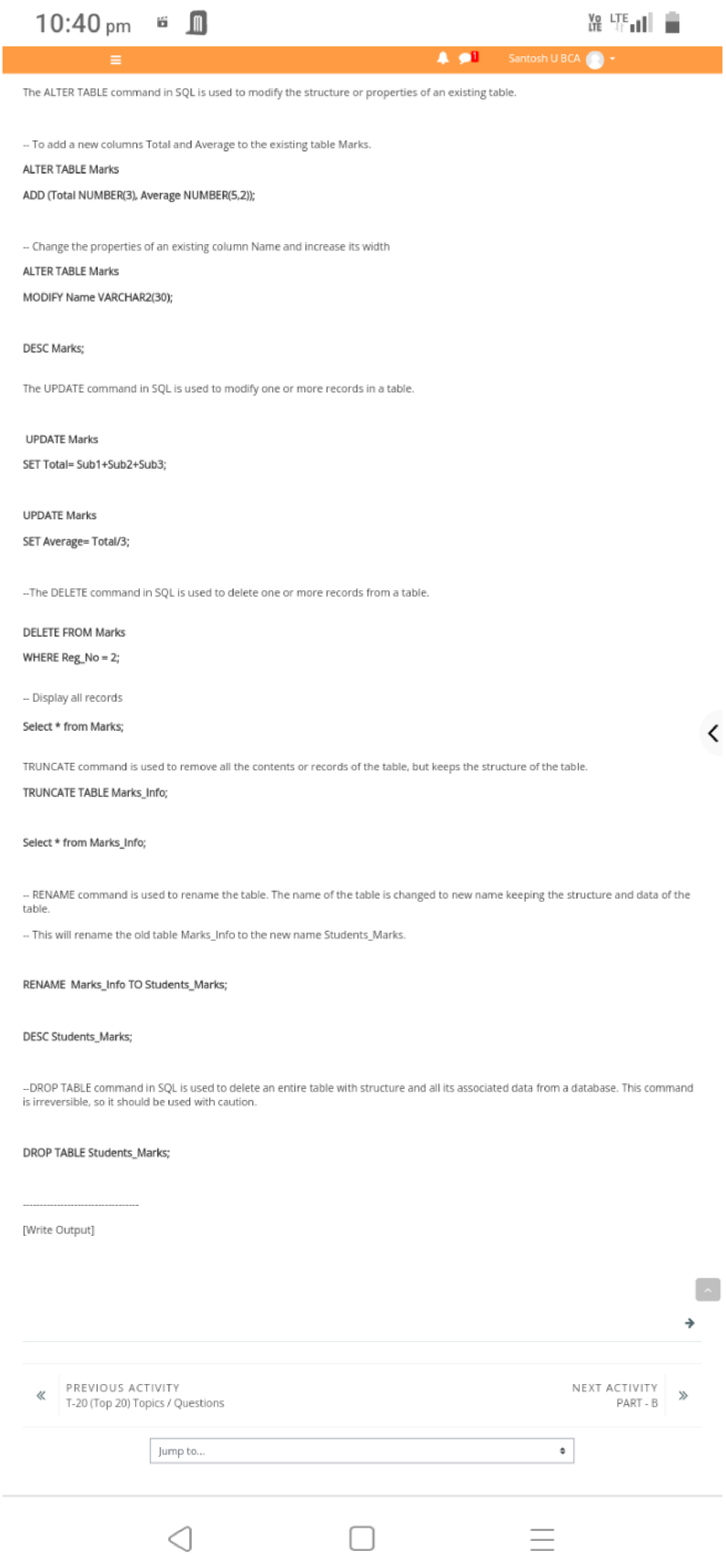
To create Marks table with Register number and marks in three subjects sub1,sub2,sub3.

```
CREATE TABLE Marks(
 Reg_No VARCHAR2(12) PRIMARY KEY,
 Name VARCHAR2(20) NOT NULL,
 Sub1 NUMBER(3),
 Sub2 NUMBER(3),
 Sub3 NUMBER(3));
```

```
-- To see the structure / description of the table

DESCRIBE Marks;

-- The INSERT INTO command in SQL is used to add new records to an existing table.

INSERT INTO Marks (Reg_No, Name, Sub1, Sub2, Sub3)
VALUES ('UG0001','Ram',88,69,79);

INSERT INTO Marks (Reg_No, Name, Sub1, Sub2, Sub3)
VALUES ('UG0004','Suma',78,66,70);

INSERT INTO Marks (Reg_No, Name, Sub1, Sub2, Sub3)
VALUES ('UG0002','Kumar',85,94,98);

INSERT INTO Marks (Reg_No, Name, Sub1, Sub2, Sub3)
VALUES ('UG0003','Ali',80,66,64);

--OR
-- To insert multiple values
INSERT INTO Marks (Reg_No, Name, Sub1, Sub2, Sub3)
VALUES ('&Reg_No','&Name',&Sub1,&Sub2,&Sub3);

Select * from Marks;

-- Create a new table called Marks_info with the same structure and data as Marks table
 Create table Marks_Info
As select * from Marks;
```

The ALTER TABLE command in SQL is used to modify the structure or properties of an existing table.

```
-- To add a new columns Total and Average to the existing table Marks.
ALTER TABLE Marks
ADD (Total NUMBER(3), Average NUMBER(5,2));
```



The ALTER TABLE command in SQL is used to modify the structure or properties of an existing table.

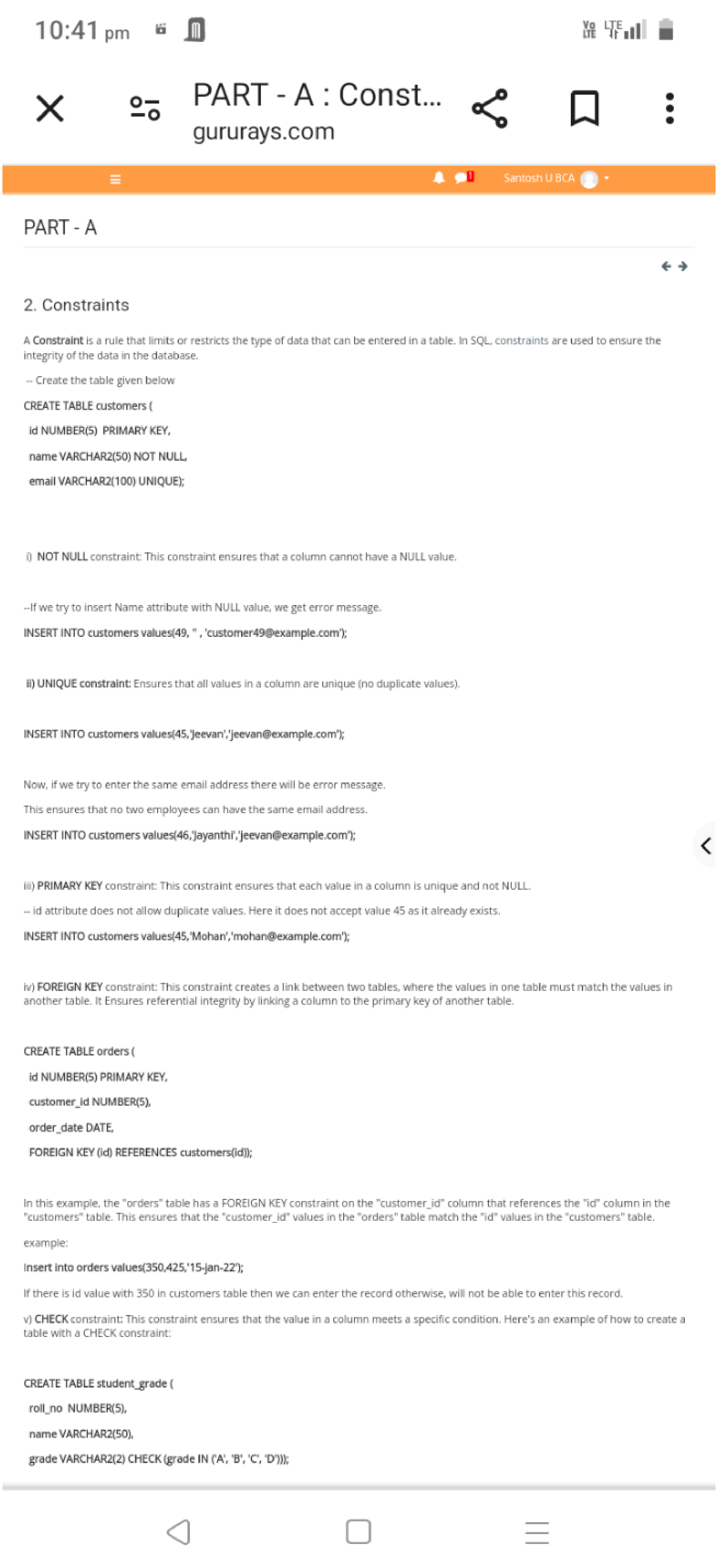
```
-- To add a new columns Total and Average to the existing table Marks.

ALTER TABLE Marks

ADD (Total NUMBER(3), Average NUMBER(5,2));


-- Change the properties of an existing column Name and increase its width

ALTER TABLE Marks

MODIFY Name VARCHAR2(30);


DESC Marks;
```

The UPDATE command in SQL is used to modify one or more records in a table.

```
 UPDATE Marks

SET Total= Sub1+Sub2+Sub3;


UPDATE Marks

SET Average= Total/3;


--The DELETE command in SQL is used to delete one or more records from a table.

DELETE FROM Marks

WHERE Reg_No = 2;

-- Display all records

Select * from Marks;
```

TRUNCATE command is used to remove all the contents or records of the table, but keeps the structure of the table.

```
TRUNCATE TABLE Marks_Info;


Select * from Marks_Info;


-- RENAME command is used to rename the table. The name of the table is changed to new name keeping the structure and data of the table.

-- This will rename the old table Marks_Info to the new name Students_Marks.


RENAME  Marks_Info TO Students_Marks;


DESC Students_Marks;


--DROP TABLE command in SQL is used to delete an entire table with structure and all its associated data from a database. This command is irreversible, so it should be used with caution.


DROP TABLE Students_Marks;


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

[Write Output]
```



PART - A

2. Constraints

A **Constraint** is a rule that limits or restricts the type of data that can be entered in a table. In SQL, constraints are used to ensure the integrity of the data in the database.

```
-- Create the table given below
CREATE TABLE customers (
 id NUMBER(5)  PRIMARY KEY,
 name VARCHAR2(50) NOT NULL,
 email VARCHAR2(100) UNIQUE);
```

i) **NOT NULL** constraint: This constraint ensures that a column cannot have a NULL value.

--If we try to insert Name attribute with NULL value, we get error message.

```
INSERT INTO customers values(49, '' , 'customer49@example.com');
```

ii) **UNIQUE constraint:** Ensures that all values in a column are unique (no duplicate values).

```
INSERT INTO customers values(45,'Jeevan','jeevan@example.com');
```

Now, if we try to enter the same email address there will be error message.

This ensures that no two employees can have the same email address.

```
INSERT INTO customers values(46,'Jayanthi','jeevan@example.com');
```

iii) **PRIMARY KEY** constraint: This constraint ensures that each value in a column is unique and not NULL.

-- id attribute does not allow duplicate values. Here it does not accept value 45 as it already exists.

```
INSERT INTO customers values(45,'Mohan','mohan@example.com');
```

iv) **FOREIGN KEY** constraint: This constraint creates a link between two tables, where the values in one table must match the values in another table. It Ensures referential integrity by linking a column to the primary key of another table.

```
CREATE TABLE orders (
 id NUMBER(5) PRIMARY KEY,
 customer_id NUMBER(5),
 order_date DATE,
 FOREIGN KEY (id) REFERENCES customers(id));
```

In this example, the "orders" table has a FOREIGN KEY constraint on the "customer_id" column that references the "id" column in the "customers" table. This ensures that the "customer_id" values in the "orders" table match the "id" values in the "customers" table.

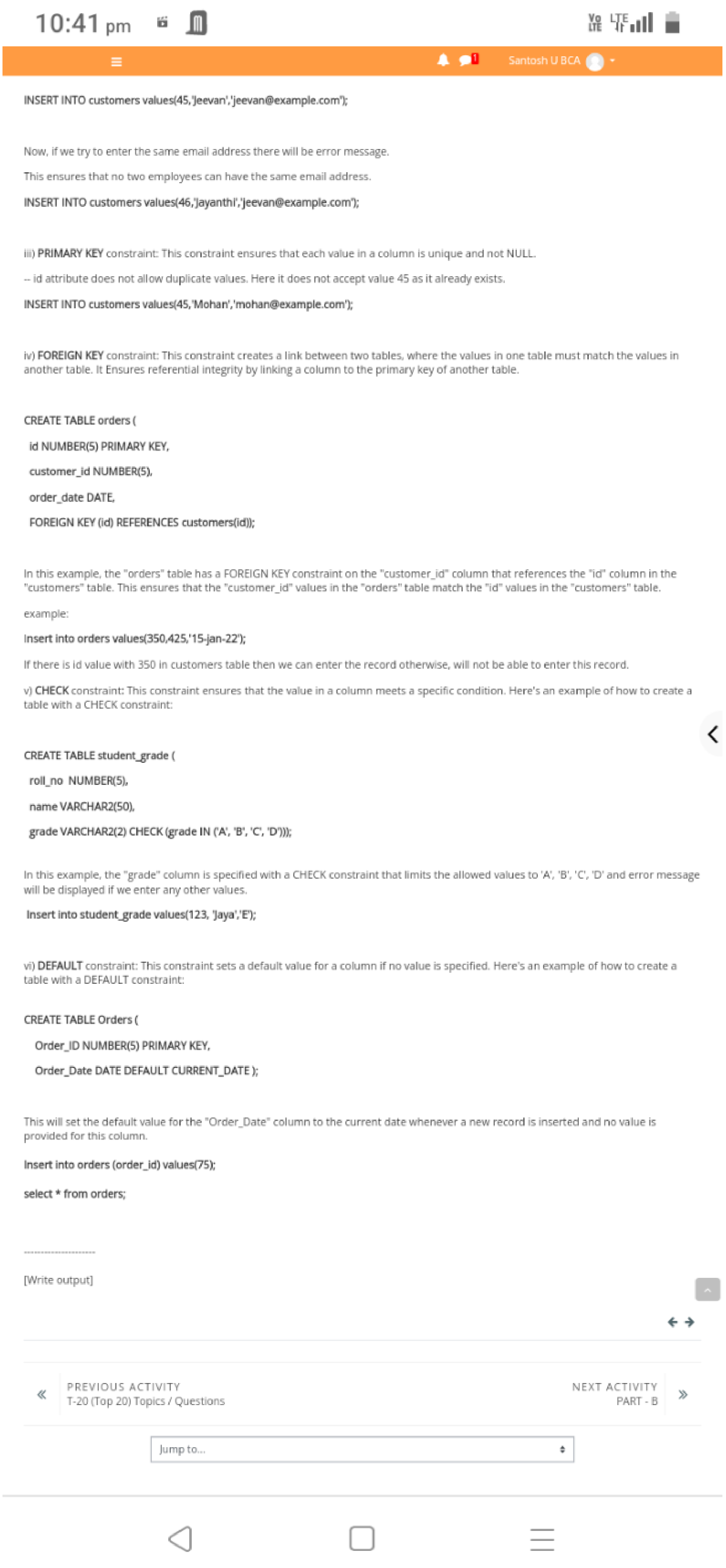
example:

```
Insert into orders values(350,425,'15-jan-22');
```

If there is id value with 350 in customers table then we can enter the record otherwise, will not be able to enter this record.

v) **CHECK** constraint: This constraint ensures that the value in a column meets a specific condition. Here's an example of how to create a table with a CHECK constraint:

```
CREATE TABLE student_grade (
 roll_no  NUMBER(5),
 name VARCHAR2(50),
 grade VARCHAR2(2) CHECK (grade IN ('A', 'B', 'C', 'D')));
```



```
INSERT INTO customers values(45,'Jeevan','jeevan@example.com');
```

Now, if we try to enter the same email address there will be error message.

This ensures that no two employees can have the same email address.

```
INSERT INTO customers values(46,'Jayanthi','jeevan@example.com');
```

iii) **PRIMARY KEY** constraint: This constraint ensures that each value in a column is unique and not NULL.

-- id attribute does not allow duplicate values. Here it does not accept value 45 as it already exists.

```
INSERT INTO customers values(45,'Mohan','mohan@example.com');
```

iv) **FOREIGN KEY** constraint: This constraint creates a link between two tables, where the values in one table must match the values in another table. It Ensures referential integrity by linking a column to the primary key of another table.

```
CREATE TABLE orders (
 id NUMBER(5) PRIMARY KEY,
 customer_id NUMBER(5),
 order_date DATE,
 FOREIGN KEY (id) REFERENCES customers(id));
```

In this example, the "orders" table has a FOREIGN KEY constraint on the "customer_id" column that references the "id" column in the "customers" table. This ensures that the "customer_id" values in the "orders" table match the "id" values in the "customers" table.

example:

```
Insert into orders values(350,425,'15-jan-22');
```

If there is id value with 350 in customers table then we can enter the record otherwise, will not be able to enter this record.

v) **CHECK** constraint: This constraint ensures that the value in a column meets a specific condition. Here's an example of how to create a table with a CHECK constraint:

```
CREATE TABLE student_grade (
 roll_no  NUMBER(5),
 name VARCHAR2(50),
 grade VARCHAR2(2) CHECK (grade IN ('A', 'B', 'C', 'D')));
```

In this example, the "grade" column is specified with a CHECK constraint that limits the allowed values to 'A', 'B', 'C', 'D' and error message will be displayed if we enter any other values.

```
 Insert into student_grade values(123, 'Jaya','E');
```

vi) **DEFAULT** constraint: This constraint sets a default value for a column if no value is specified. Here's an example of how to create a table with a DEFAULT constraint:

```
CREATE TABLE Orders (
   Order_ID NUMBER(5) PRIMARY KEY,
   Order_Date DATE DEFAULT CURRENT_DATE );
```

This will set the default value for the "Order_Date" column to the current date whenever a new record is inserted and no value is provided for this column.

```
Insert into orders (order_id) values(75);

select * from orders;
```

[Write output]

PART - A

3. Implementation of JOINS

Different types of joins can be used to combine data from two or more tables.

First create required tables with sample data

```
-- Create the Orders table
CREATE TABLE Orders
 ( OrderID NUMBER(5),
   CustomerID NUMBER(5),
   OrderName VARCHAR2(50));

-- Insert data into the Orders table
INSERT INTO Orders VALUES (1, 1, 'Order1');
 INSERT INTO Orders VALUES(2, 2, 'Order2');
 INSERT INTO Orders VALUES (3, 3, 'Order3');
INSERT INTO Orders VALUES(4, 5, 'Order4');

-- Create the Customers table
CREATE TABLE Customers
  ( CustomerID NUMBER(5),
   CustomerName VARCHAR2(5),
   Contact VARCHAR2(50));

-- Insert data into the Customers table
INSERT INTO Customers VALUES (1, 'Customer1', 'Contact1');
INSERT INTO Customers VALUES (2, 'Customer2', 'Contact2');
INSERT INTO Customers VALUES (3, 'Customer3', 'Contact3');
INSERT INTO Customers VALUES (4, 'Customer4', 'Contact4');
```

INNER JOIN: This type of join returns only the matching rows from both tables.

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderName
FROM Orders, Customers
WHERE Orders.CustomerID = Customers.CustomerID;
```

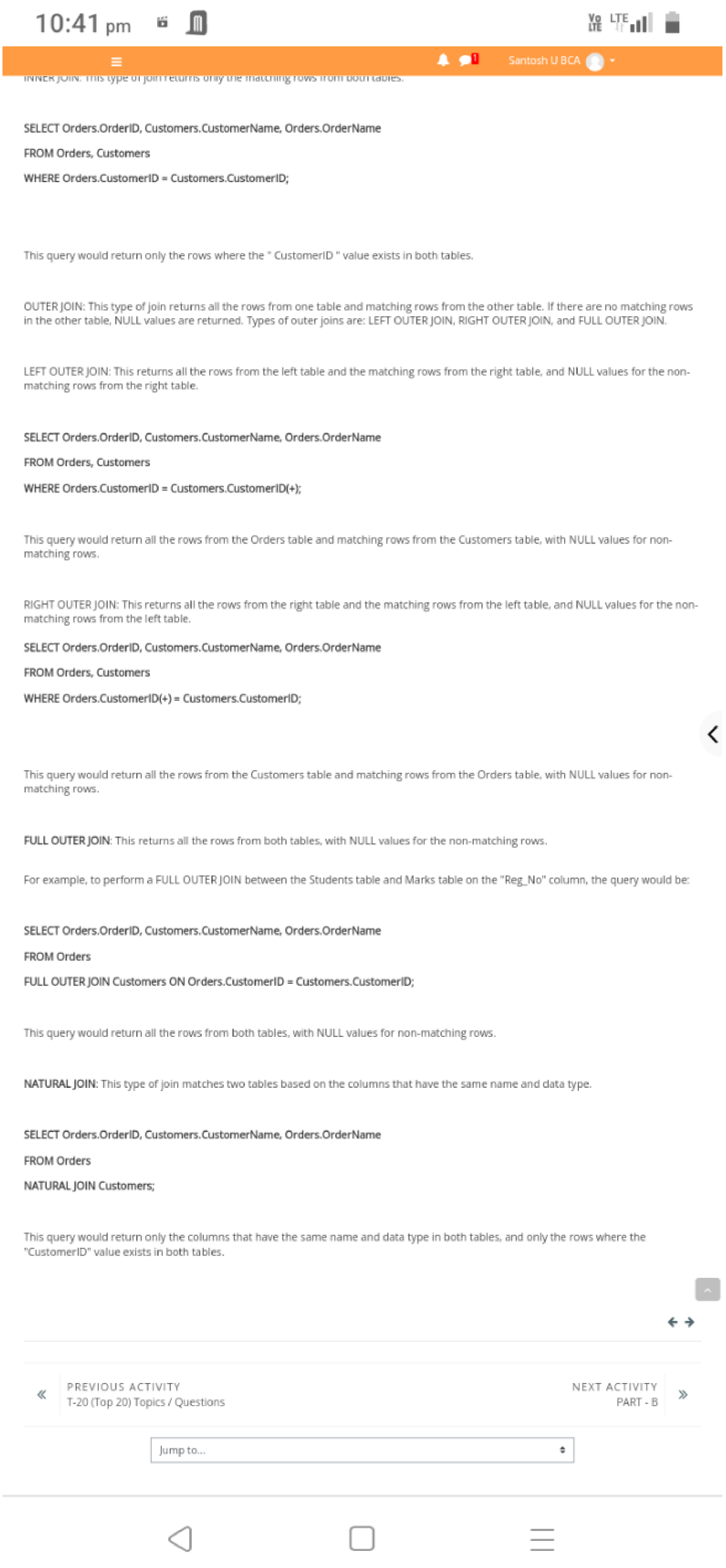
This query would return only the rows where the " CustomerID " value exists in both tables.

OUTER JOIN: This type of join returns all the rows from one table and matching rows from the other table. If there are no matching rows in the other table, NULL values are returned. Types of outer joins are: LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN.

LEFT OUTER JOIN: This returns all the rows from the left table and the matching rows from the right table, and NULL values for the non-matching rows from the right table.

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderName
FROM Orders, Customers
WHERE Orders.CustomerID = Customers.CustomerID(+);
```

This query would return all the rows from the Orders table and matching rows from the Customers table, with NULL values for non-matching rows.



INNER JOIN: This type of join returns only the matching rows from both tables.

SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderName

FROM Orders, Customers

WHERE Orders.CustomerID = Customers.CustomerID;

This query would return only the rows where the " CustomerID " value exists in both tables.

OUTER JOIN: This type of join returns all the rows from one table and matching rows from the other table. If there are no matching rows in the other table, NULL values are returned. Types of outer joins are: LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN.

LEFT OUTER JOIN: This returns all the rows from the left table and the matching rows from the right table, and NULL values for the non-matching rows from the right table.

SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderName

FROM Orders, Customers

WHERE Orders.CustomerID = Customers.CustomerID(+);

This query would return all the rows from the Orders table and matching rows from the Customers table, with NULL values for non-matching rows.

RIGHT OUTER JOIN: This returns all the rows from the right table and the matching rows from the left table, and NULL values for the non-matching rows from the left table.

SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderName

FROM Orders, Customers

WHERE Orders.CustomerID(+) = Customers.CustomerID;

This query would return all the rows from the Customers table and matching rows from the Orders table, with NULL values for non-matching rows.

FULL OUTER JOIN: This returns all the rows from both tables, with NULL values for the non-matching rows.

For example, to perform a FULL OUTER JOIN between the Students table and Marks table on the "Reg_No" column, the query would be:

SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderName

FROM Orders

FULL OUTER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;

This query would return all the rows from both tables, with NULL values for non-matching rows.

NATURAL JOIN: This type of join matches two tables based on the columns that have the same name and data type.

SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderName

FROM Orders

NATURAL JOIN Customers;

This query would return only the columns that have the same name and data type in both tables, and only the rows where the "CustomerID" value exists in both tables.

« PREVIOUS ACTIVITY
T-20 (Top 20) Topics / Questions

NEXT ACTIVITY »
PART - B

Jump to...

PART - A

4. Grouping and Sorting

Ex. 4.

i) Group By and having ii) Order By

--First Create required table.

```
CREATE TABLE employees

( emp_no NUMBER(5), name VARCHAR2(50), age NUMBER(3), department VARCHAR2(50), salary NUMBER(10,2) );
```

-- Insert sample records

```
INSERT INTO employees VALUES (115,'Rajani',27,'Accounts',46000.00);

INSERT INTO employees VALUES (115,'Rajani',29,'Accounts',55000.00);

INSERT INTO employees VALUES (115,'Rajani',32,'Sales',43000.00) ;

INSERT INTO employees VALUES (115,'Rajani',30,'Computer',65000.00);

INSERT INTO employees VALUES (115,'Rajani',31,'Sales',58000.00);
```

i)

The GROUP BY clause is used in SQL to group the results of a query by one or more columns. It is typically used in conjunction with aggregate functions like COUNT, SUM, AVG, etc. to group the data based on specific criteria.

```
SELECT department, COUNT(*) as num_employees

FROM employees

GROUP BY department;
```

In this example, we are selecting the department column and counting the number of employees in each department using the COUNT function. We are then grouping the results by the department column using the GROUP BY clause.

The HAVING clause is used in SQL to filter the results of a GROUP BY query based on a condition. It is similar to the WHERE clause, but is used to filter groups of rows rather than individual rows. Here is an example of how the HAVING clause can be used in SQL:

```
SELECT department, AVG(salary) as avg_salary

FROM employees

GROUP BY department

HAVING AVG(salary) > 50000;
```

In this example, we are selecting the department column and calculating the average salary of employees in each department with average salary more that 50000 (using the AVG function).

ii) Using ORDER BY

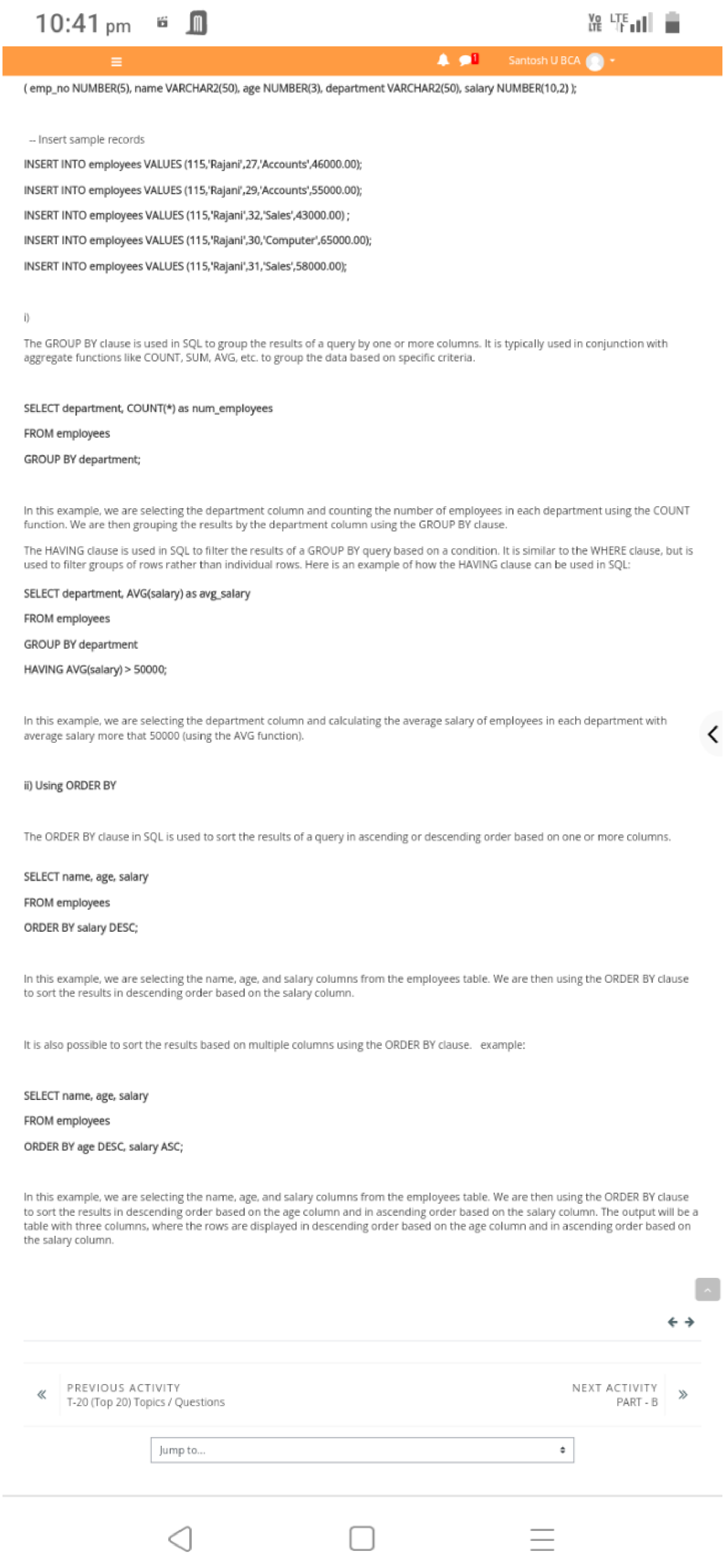
The ORDER BY clause in SQL is used to sort the results of a query in ascending or descending order based on one or more columns.

```
SELECT name, age, salary

FROM employees

ORDER BY salary DESC;
```

In this example, we are selecting the name, age, and salary columns from the employees table. We are then using the ORDER BY clause to sort the results in descending order based on the salary column.

It is also possible to sort the results based on multiple columns using the ORDER BY clause. example:

```
SELECT name, age, salary

FROM employees
```



```
( emp_no NUMBER(5), name VARCHAR2(50), age NUMBER(3), department VARCHAR2(50), salary NUMBER(10,2) );

-- Insert sample records
INSERT INTO employees VALUES (115,'Rajani',27,'Accounts',46000.00);
INSERT INTO employees VALUES (115,'Rajani',29,'Accounts',55000.00);
INSERT INTO employees VALUES (115,'Rajani',32,'Sales',43000.00) ;
INSERT INTO employees VALUES (115,'Rajani',30,'Computer',65000.00);
INSERT INTO employees VALUES (115,'Rajani',31,'Sales',58000.00);
```

i)

The GROUP BY clause is used in SQL to group the results of a query by one or more columns. It is typically used in conjunction with aggregate functions like COUNT, SUM, AVG, etc. to group the data based on specific criteria.

```
SELECT department, COUNT(*) as num_employees
FROM employees
GROUP BY department;
```

In this example, we are selecting the department column and counting the number of employees in each department using the COUNT function. We are then grouping the results by the department column using the GROUP BY clause.

The HAVING clause is used in SQL to filter the results of a GROUP BY query based on a condition. It is similar to the WHERE clause, but is used to filter groups of rows rather than individual rows. Here is an example of how the HAVING clause can be used in SQL:

```
SELECT department, AVG(salary) as avg_salary
FROM employees
GROUP BY department
HAVING AVG(salary) > 50000;
```

In this example, we are selecting the department column and calculating the average salary of employees in each department with average salary more that 50000 (using the AVG function).

ii) Using ORDER BY

The ORDER BY clause in SQL is used to sort the results of a query in ascending or descending order based on one or more columns.

```
SELECT name, age, salary
FROM employees
ORDER BY salary DESC;
```

In this example, we are selecting the name, age, and salary columns from the employees table. We are then using the ORDER BY clause to sort the results in descending order based on the salary column.

It is also possible to sort the results based on multiple columns using the ORDER BY clause.   example:

```
SELECT name, age, salary
FROM employees
ORDER BY age DESC, salary ASC;
```

In this example, we are selecting the name, age, and salary columns from the employees table. We are then using the ORDER BY clause to sort the results in descending order based on the age column and in ascending order based on the salary column. The output will be a table with three columns, where the rows are displayed in descending order based on the age column and in ascending order based on the salary column.

PART - A

5. Views

Ex. 5.

In SQL, a view is a virtual table based on the result set of a SELECT statement. A view can be used to simplify complex queries, restrict access to sensitive data, and provide a level of abstraction between the underlying data and the user. They can be created from single or multiple tables, used in queries to retrieve data, updated to include new columns or calculations, and dropped when they are no longer needed.

```
--First Create required tables.

CREATE TABLE employees

( emp_id NUMBER(5), name VARCHAR2(25), department_id  NUMBER(5), salary NUMBER(10,2) );


CREATE TABLE departments

( department_id  NUMBER(5)  PRIMARY KEY, dept_name  VARCHAR2(25)  NOT NULL);


-- Insert sample records

SQL> INSERT INTO employees values(123,'Vidya',5,56000.00);

SQL> INSERT INTO employees values(234,'Vijaya',8,45000.00);

SQL> INSERT INTO employees values(456,'Sumana',3,56000.00);


SQL> INSERT INTO departments values(3,'Accounts');

SQL> INSERT INTO departments values(5,'Accounts');

SQL> INSERT INTO departments values(8,'Marketing');
```

i) Creating a view:

```
CREATE VIEW emp_view AS

SELECT emp_id, name, salary

FROM employees

WHERE salary > 50000;
```

In this example, we are creating a view called emp_view that selects the columns emp_id, name, and salary from the employees table and restricts the result set to only include employees with a salary greater than 50000.

Once the view is created, we can query it just like any other table.

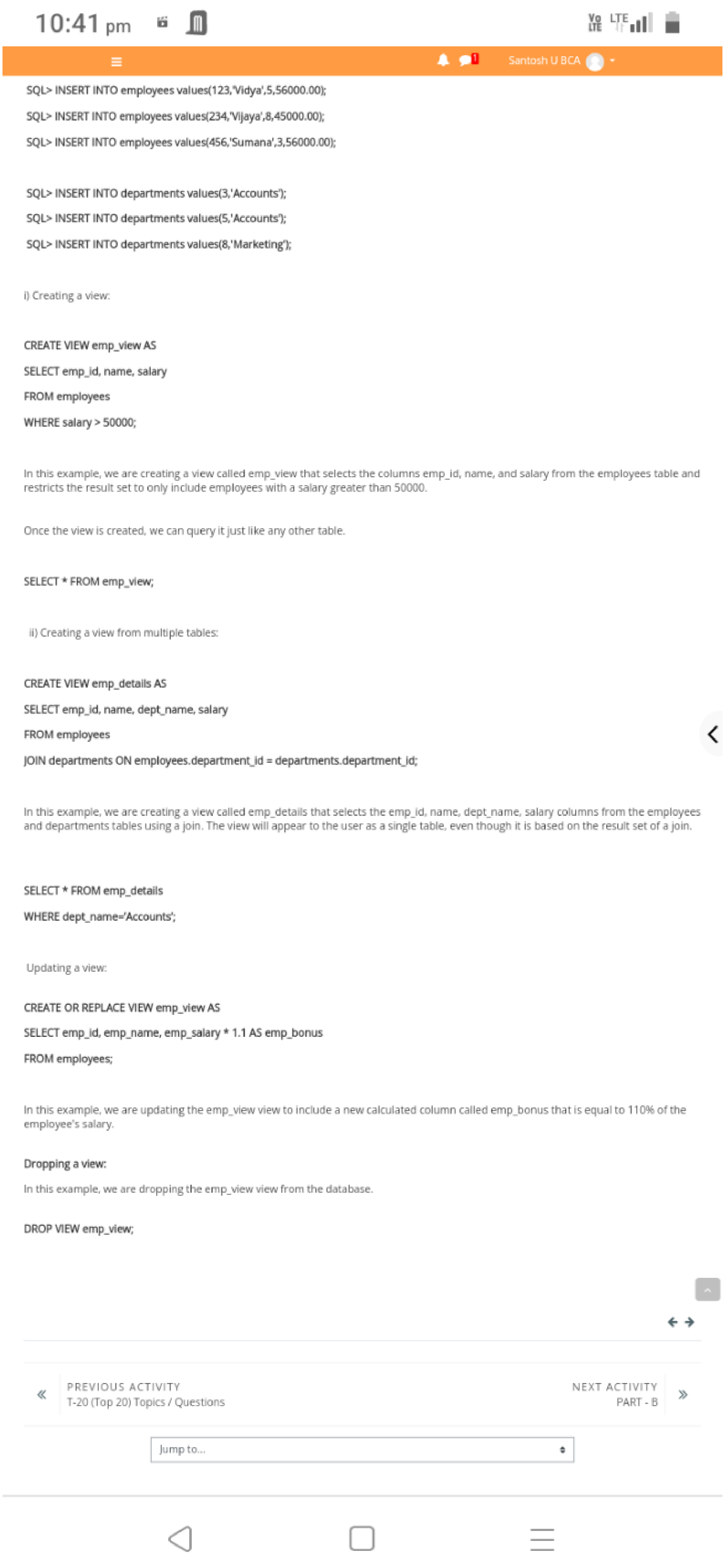
```
SELECT * FROM emp_view;
```

ii) Creating a view from multiple tables:

```
CREATE VIEW emp_details AS

SELECT emp_id, name, dept_name, salary

FROM employees

JOIN departments ON employees.department_id = departments.department_id;
```

In this example, we are creating a view called emp_details that selects the emp_id, name, dept_name, salary columns from the employees and departments tables using a join. The view will appear to the user as a single table, even though it is based on the result set of a join.

```
SELECT * FROM emp_details

WHERE dept_name='Accounts';
```



```
SQL> INSERT INTO employees values(123,'Vidya',5,56000.00);
SQL> INSERT INTO employees values(234,'Vijaya',8,45000.00);
SQL> INSERT INTO employees values(456,'Sumana',3,56000.00);

SQL> INSERT INTO departments values(3,'Accounts');
SQL> INSERT INTO departments values(5,'Accounts');
SQL> INSERT INTO departments values(8,'Marketing');
```

i) Creating a view:

```
CREATE VIEW emp_view AS
SELECT emp_id, name, salary
FROM employees
WHERE salary > 50000;
```

In this example, we are creating a view called emp_view that selects the columns emp_id, name, and salary from the employees table and restricts the result set to only include employees with a salary greater than 50000.

Once the view is created, we can query it just like any other table.

```
SELECT * FROM emp_view;
```

ii) Creating a view from multiple tables:

```
CREATE VIEW emp_details AS
SELECT emp_id, name, dept_name, salary
FROM employees
JOIN departments ON employees.department_id = departments.department_id;
```

In this example, we are creating a view called emp_details that selects the emp_id, name, dept_name, salary columns from the employees and departments tables using a join. The view will appear to the user as a single table, even though it is based on the result set of a join.

```
SELECT * FROM emp_details
WHERE dept_name='Accounts';
```

Updating a view:

```
CREATE OR REPLACE VIEW emp_view AS
SELECT emp_id, emp_name, emp_salary * 1.1 AS emp_bonus
FROM employees;
```

In this example, we are updating the emp_view view to include a new calculated column called emp_bonus that is equal to 110% of the employee's salary.

Dropping a view:

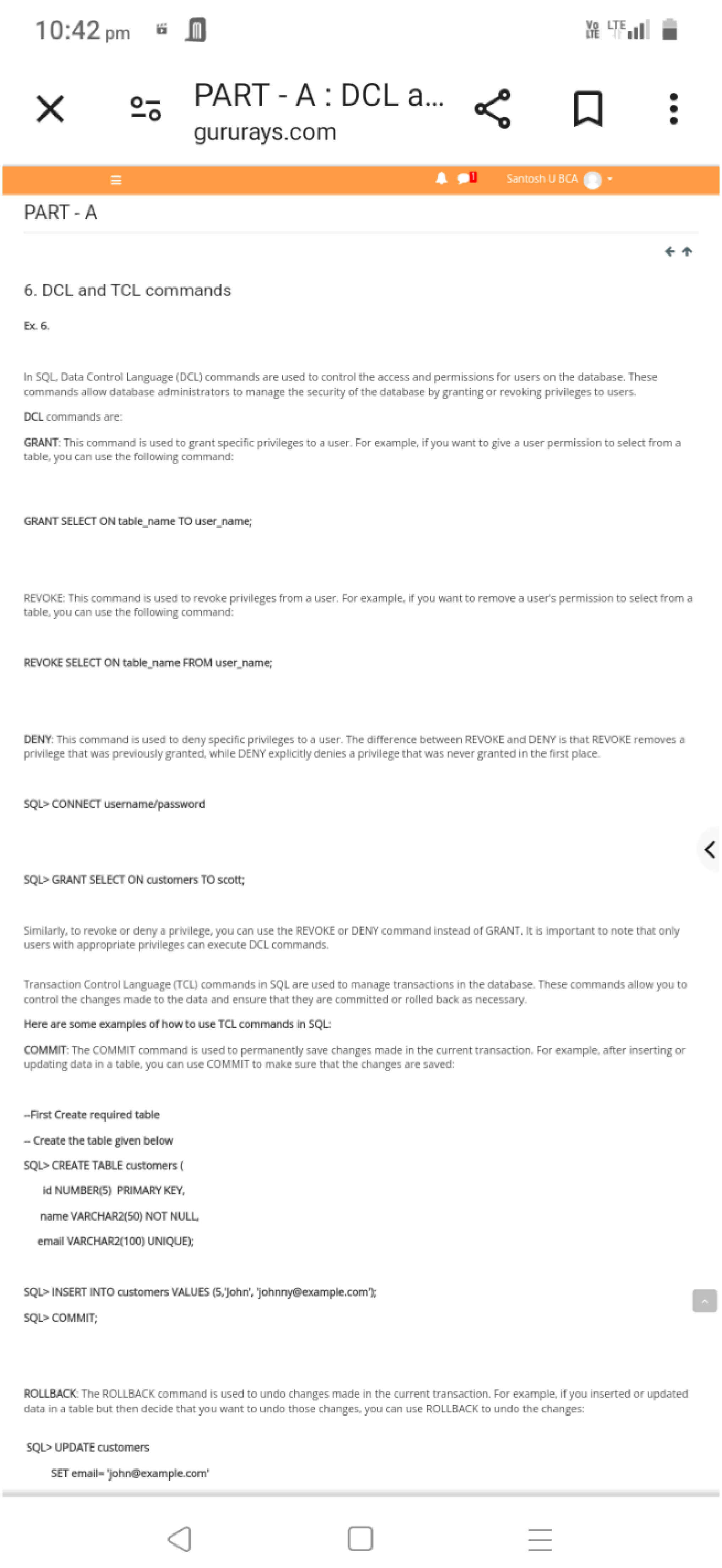
In this example, we are dropping the emp_view view from the database.

```
DROP VIEW emp_view;
```

« PREVIOUS ACTIVITY
T-20 (Top 20) Topics / Questions

NEXT ACTIVITY »
PART - B

Jump to...



PART - A

6. DCL and TCL commands

Ex. 6.

In SQL, Data Control Language (DCL) commands are used to control the access and permissions for users on the database. These commands allow database administrators to manage the security of the database by granting or revoking privileges to users.

DCL commands are:

GRANT: This command is used to grant specific privileges to a user. For example, if you want to give a user permission to select from a table, you can use the following command:

```
GRANT SELECT ON table_name TO user_name;
```

REVOKE: This command is used to revoke privileges from a user. For example, if you want to remove a user's permission to select from a table, you can use the following command:

```
REVOKE SELECT ON table_name FROM user_name;
```

DENY: This command is used to deny specific privileges to a user. The difference between REVOKE and DENY is that REVOKE removes a privilege that was previously granted, while DENY explicitly denies a privilege that was never granted in the first place.

```
SQL> CONNECT username/password

SQL> GRANT SELECT ON customers TO scott;
```

Similarly, to revoke or deny a privilege, you can use the REVOKE or DENY command instead of GRANT. It is important to note that only users with appropriate privileges can execute DCL commands.

Transaction Control Language (TCL) commands in SQL are used to manage transactions in the database. These commands allow you to control the changes made to the data and ensure that they are committed or rolled back as necessary.

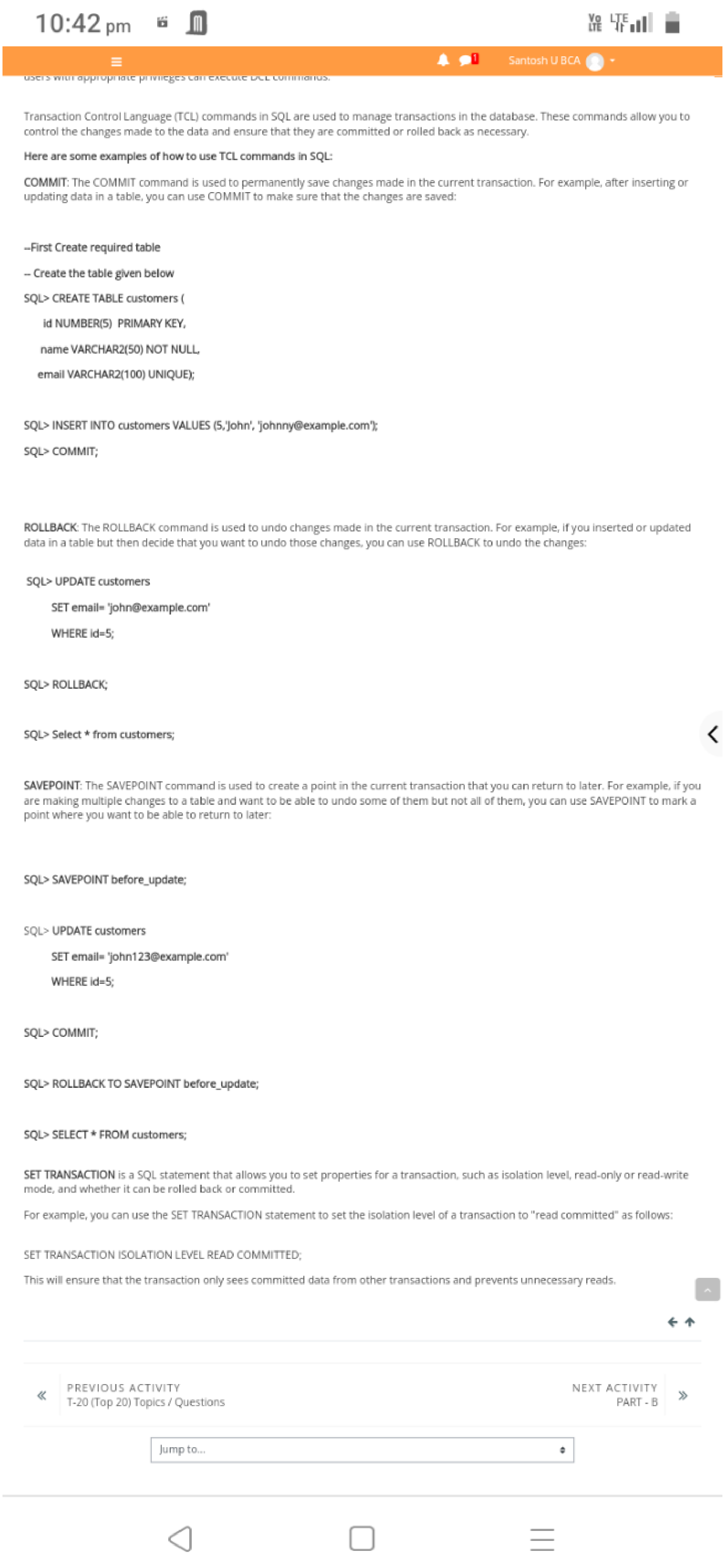
Here are some examples of how to use TCL commands in SQL:

COMMIT: The COMMIT command is used to permanently save changes made in the current transaction. For example, after inserting or updating data in a table, you can use COMMIT to make sure that the changes are saved:

```
--First Create required table
-- Create the table given below
SQL> CREATE TABLE customers (
    id NUMBER(5)  PRIMARY KEY,
   name VARCHAR2(50) NOT NULL,
   email VARCHAR2(100) UNIQUE);

SQL> INSERT INTO customers VALUES (5,'John', 'johnny@example.com');
SQL> COMMIT;
```

ROLLBACK: The ROLLBACK command is used to undo changes made in the current transaction. For example, if you inserted or updated data in a table but then decide that you want to undo those changes, you can use ROLLBACK to undo the changes:

```
 SQL> UPDATE customers
      SET email= 'john@example.com'
```


10:42 pm

Santosh U BCA

users with appropriate privileges can execute DCL commands.

Transaction Control Language (TCL) commands in SQL are used to manage transactions in the database. These commands allow you to control the changes made to the data and ensure that they are committed or rolled back as necessary.

Here are some examples of how to use TCL commands in SQL:

COMMIT: The COMMIT command is used to permanently save changes made in the current transaction. For example, after inserting or updating data in a table, you can use COMMIT to make sure that the changes are saved:

```
--First Create required table
-- Create the table given below
SQL> CREATE TABLE customers (
   id NUMBER(5)  PRIMARY KEY,
  name VARCHAR2(50) NOT NULL,
 email VARCHAR2(100) UNIQUE);

SQL> INSERT INTO customers VALUES (5,'John', 'johnny@example.com');
SQL> COMMIT;
```

ROLLBACK: The ROLLBACK command is used to undo changes made in the current transaction. For example, if you inserted or updated data in a table but then decide that you want to undo those changes, you can use ROLLBACK to undo the changes:

```
 SQL> UPDATE customers
      SET email= 'john@example.com'
      WHERE id=5;

SQL> ROLLBACK;

SQL> Select * from customers;
```

SAVEPOINT: The SAVEPOINT command is used to create a point in the current transaction that you can return to later. For example, if you are making multiple changes to a table and want to be able to undo some of them but not all of them, you can use SAVEPOINT to mark a point where you want to be able to return to later:

```
SQL> SAVEPOINT before_update;

SQL> UPDATE customers
     SET email= 'john123@example.com'
     WHERE id=5;

SQL> COMMIT;

SQL> ROLLBACK TO SAVEPOINT before_update;

SQL> SELECT * FROM customers;
```

SET TRANSACTION is a SQL statement that allows you to set properties for a transaction, such as isolation level, read-only or read-write mode, and whether it can be rolled back or committed.

For example, you can use the SET TRANSACTION statement to set the isolation level of a transaction to "read committed" as follows:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

This will ensure that the transaction only sees committed data from other transactions and prevents unnecessary reads.

« PREVIOUS ACTIVITY
T-20 (Top 20) Topics / Questions

NEXT ACTIVITY »
PART - B

Jump to...