

GTU Important questions (RDBMS)

1. Explain View with example.

➤ A **view** is a virtual or logical table that allows to view or manipulate the parts of the tables.

➤ A view is derived from one or more tables known as **base tables**.

Types of Views

➤ View can be classified into two categories based on which type of operations they allow:

1) Read-only View:

o Allows only **SELECT** operation, this means user can only view data.

o No **INSERT, UPDATE or DELETE** operations are allowed. This means contents of base table cannot be modified.

2) Updateable View:

o Allows **SELECT** as well as **INSERT, UPDATE and DELETE** operations.

o This means contents of the base tables can be displayed as well as modified.

➤ Creating a View

A view can be created using syntax as given below:

Syntax:

CREATE [OR REPLACE] VIEW viewName

As SELECT

[WITH READ ONLY];

➤ This statement creates a view based on query specified in **SELECT** statement.

- **OR REPLACE** option re-creates the view if it is already existing maintaining the privileges granted to view that is given by viewName.
- **WITH READ ONLY** option creates **read-only views**. If this option is not provided then **by default updatable views** are created.
- **Example:**

```
CREATE VIEW Acc_vvn
AS SELECT * FROM Account
WHERE bname = 'vvn';
```

2. Explain GRANT command by example.

- **GRANT** command is used to granting privileges means to give permission to some user to access database object or a part of a database object.
- This command provides various types of access to database object such as tables, views and sequences.

➤ Syntax:

```
GRANT object privileges
ON object name
TO user name
[ WITH GRANT OPTION ];
```

- The owner of a database object can grant all privileges or specific privileges to other users.
- The **WITH GRANT OPTION** allows the grantee. User to which privilege is granted to in turn grant object privilege to other users.
- User can grant all or specific privileges owned by him/her.

Privilege	Allows user...
ALL	To perform all the operation listed below.
ALTER	To change the table structure using ALTER command.
DELETE	To delete records from the table using DELETE command.
INDEX	To create an index on the table using CREATE INDEX

	command.
INSERT	To insert records into the table using INSERT INTO command.
REFERENCES	To reference table while creating foreign keys.
SELECT	To query the table using SELECT command.
UPDATE	To modify the records in the table using UPDATE command.

➤ **Example:**

```
GRANT ALL
ON      Customer
TO      user2
WITH GRANT OPTION;
```

3. Write short note: Explicit locks

- User can lock data in a table on its own instead of automatic locking provided by Oracle. These types of locks are called **Explicit locks**.
- An owner of a table can place an explicit lock on the table. Some other users can also place an explicit lock if they have privilege.
- An explicit lock always overrides the implicit locks placed by oracle on its own.
- An entire table or records of the table can be explicitly locked by using one of these two commands:

1) The **SELECT FOR UPDATE** Statement

➤ **Syntax:**

```
SELECT * FROM tableName FOR UPDATE [ NOWAIT ];
```

- This statement is used **to acquire exclusive locks for performing updates on records**.
- Based on **WHERE** clause used with **SELECT** statement level of lock will be applied.
- If table is already locked by other user then this command simply **waits** until that lock is released.
- But, if **NOWAIT** is specified and table is not free, this command will return with an error message indicates “**Resource is Busy**”.
- Lock will be released on executing **COMMIT** or **ROLLBACK**.

➤ **Example:**

SELECT * FROM Account WHERE bname = 'vvn' FOR UPDATE;

4. Draw and explain generic PL/SQL block.

- PL/SQL code is grouped into structures called **block**.
- A block is called a **named block**, if it is given particular name to identify.
- A block is called an **anonymous block**, if it is not given any name.
- **Named blocks** are created while creating **database objects** such as function, procedure, package and trigger.
- **The structure of a typical PL/SQL block can be given as below:**

```

DECLARE      -- Optional
               <Declaration Section>
BEGIN        -- Mandatory
               <Executable Commands>
EXCEPTION   -- Optional
               <Exception Handling>
END ;        -- Mandatory

```

- Notice that **DECLARE** and **EXCEPTION** are **optional** while **BEGIN** and **END** are **mandatory**.
- Also, ‘;’ at the end to **terminate the block**.
- A block of PL/SQL code contains three sections given as below:
 - **Declarations**
 - **Executable Commands**
 - **Exception Handling**
- **Declarations:**
 - This section starts with the keyword ‘**DECLARE**’.
 - It defines and initializes **variables** and **cursors** used in the block.
 - This section is **optional section**. If block does not require any variables or cursors then this section can be omitted from the block.

➤ **Executable Commands:**

- This section starts with the keyword '**BEGIN**'.
- This is the only **mandatory section** in the PL/SQL block.
- It contains various SQL and PL/SQL statements providing functionalities like **data retrieval, manipulation, looping and branching**.

➤ **Exception Handling:**

- This section starts with the keyword '**EXCEPTION**'.
- This section handles errors that arise during the **execution of data manipulation statements** in '**executable commands**' section.
- This section is **optional section**. If block does not handle any exception explicitly then this section can be omitted from the block.

5. Give advantages of PL/SQL.

❖ **Procedural Capabilities:**

- PL/SQL provides procedural capabilities such as **condition checking, branching and looping**.
- This enables programmer to control execution of a program based on some conditions and user inputs.

❖ **Support to variables:**

- PL/SQL supports declaration and use of variables.
- These variables can be used to store intermediate results of a query or some expression.

❖ **Error Handling:**

- When an error occurs, user friendly message can be displayed.
- Also, execution of program can be controlled instead of abruptly terminating the program.

❖ **User Defined Functions:**

- Along with a large set of in-build functions, PL/SQL also supports **user defined functions** and **procedures**.

❖ **Portability:**

- Programs written in PL/SQL are portable.
- It means, programs can be **transferred and executed** from any other computer hardware and operating system, where Oracle is operational.

❖ **Sharing of Code:**

- PL/SQL allows user to store compiled code in database. This code can be **accessed and shared by different applications**.
- This code can be executed by other programming language like JAVA.

6. Explain steps to manage explicit cursors.

- Following steps required to manage an explicit cursor:
 - Declare a cursor
 - Open a cursor
 - Fetching data
 - Processing data
 - Closing cursor

❖ **Declare a Cursor:**

➤ **Syntax:**

CURSOR cursorName **IS SELECT** ;

- A cursor with cursorName is declared.
- It is mapped to a query given by SELECT statement.
- Here, only cursor will be declared. No any memory is allocated yet.

Example:

CURSOR cursorAcc **IS**

SELECT Acc_No, Balance, B_Name **FROM** Account ;

❖ **Open a Cursor:**

- Once cursor is declared we can open it.
- When cursor is opened following operations are performed:
 - Memory is allocated to store the data.
 - Execute SELECT statement associated with cursor.
 - Create active data set by retrieving data from table.
 - Set the cursor row pointer to point to first record in active data set.

Syntax:

OPEN cursorName ;

❖ **Fetching Data:**

- We cannot process selected row directly. We have to **fetch column values** of a row into **memory variables**.
- This is done by FETCH statement.
- **Syntax:**

FETCH cursorName **INTO** variable1, variable2;
- Retrieve data from the current row in the active data set and stores them in given variables.
- Data from a **single row** are fetched at a time.
- After fetching data, **updates row pointer** to point the **next row** in an active data set.
- **Variables** should be **compatible** with the columns specified in the SELECT statement.
- **Example:**

FETCH cursorAcc **INTO** no, bal, bn;

- Fetched account number, balance and branch name from **current row** in active data set and **store** them in respective variables.
- To process **more than one record**, the **FETCH** statement is enclosed within loop like **LOOP ... END LOOP** can be used.

❖ **Processing data:**

- This step involves actual processing of current row by using PL/SQL as well as SQL statements..

❖ **Closing Cursor:**

- A cursor should be closed after the processing of data completes. Once you close the cursor it will release memory allocated for that cursor.
- If user forgets to close the cursor, it will be automatically closed after termination of the program.
- **Syntax:**
CLOSE cursorName ;

7. Explain different attributes of cursor.

Attribute Name	Description
SQL%ISOPEN	If explicit cursor is open, returns TRUE . Else Return False .
SQL%FOUND	If record was fetched successfully in last FETCH statement then return TRUE . Else returns FALSE indicating no more records available in active data set.
SQL%NOTFOUND	If record was not fetched successfully in last FETCH statement returns TRUE . Else returns FALSE .
SQL%ROWCOUNT	Returns number of records fetched from active data set. It is set to ZERO when cursor is opened .
SQL%ISOPEN	If explicit cursor is open, returns TRUE . Else Return False .
SQL%FOUND	If record was fetched successfully in last FETCH statement then return TRUE . Else returns FALSE indicating no more records available in active

8. Explain Types of Triggers using Examples.

➤ Triggers can be classified based on two different criteria:

1) Based on number of times trigger action is executed.

- a) Row Trigger
- b) Statement Trigger

2) Based on timing when trigger action is executed.

- a) Before Trigger
- b) After Trigger

1) Based on number of times trigger action is executed:

Row Trigger	Statement Trigger
Fired each time the table is affected by the triggering statement.	Fired only once .
Example: If an UPDATE statement updates	Example: If an UPDATE statement updates
multiple rows of a table, a row trigger is fired once for each row affected by the	multiple rows of a table, statement trigger is fired only once .
UPDATE statement If no rows are affected by the triggering statement, a trigger will not be executed .	Trigger will be executed once , if no rows are affected by the triggering statement.

2) Based on timing when trigger action is executed:

Before Trigger	After Trigger
Trigger is executed before the triggering statement.	Trigger is executed after the triggering statement.
Used to determines whether the triggering statement should be allowed to execute or not.	Used when there is a need for a triggering statement to complete execution before trigger.

➤ These types are used in combination, provides total four types of triggers:

- 1) **Before Statement:** Execute trigger once before triggering statement.
- 2) **Before Row:** Execute trigger multiple times before triggering statement.
- 3) **After Statement:** Execute trigger once after triggering statement.
- 4) **After Row:** Execute trigger multiple times after triggering statement.

Example:

```
CREATE OR REPLACE TRIGGER balNegative
  BEFORE INSERT
  ON Account FOR EACH ROW
BEGIN
  IF      :NEW.Balance < 0 THEN
    dbms_output.put_line ('Balance is negative..');
  END IF ;
END ;
/
```

**9. Define package. Write steps to create package in PL/SQL (OR)
Structure of Package in SQL.**

- A package is one kind of database object.
- It is used to group together **logically related objects** like variables, constants, cursors, exceptions, procedures and functions.
- A successfully compiled package is stored in oracle database like procedures and functions.
- Unlike procedure and functions, **package itself cannot be called.**

Structure of a Package

- A package contains two sections:
 - 1) **Package Specification**
 - 2) **Package Body**
- **While creating packages, package specification and package body are created separately.**

1) Package Specification:

- Various objects (like variables, constants etc..) to be held by package are declared in this section.
- This declaration is global to the package, means accessible from anywhere in the package.

Syntax:

```

CREATE OR REPLACE PACKAGE packageName
IS
    Package Specification
END packageName;

```

- Package specification consists of list of variables, constants, functions, procedures and cursors.

□ Example:

```

CREATE OR REPLACE PACKAGE transaction

IS
    PROCEDURE debitAcc ( no IN Account.Acc_No%TYPE, amount IN
    NUMBER );
    FUNCTION getBalance (no IN Account.Acc_No%TYPE) RETURN
    NUMBER;
END transaction ;
/

```

2) Package Body:

- It contains the formal definition of all the objects declared in the specification section.

Syntax:

```

CREATE OR REPLACE PACKAGE BODY packageName
IS
    package body
END packageName;

```

- If a package contains only **variables, constants and exceptions** then package body is **optional**.

**10. List out any four pre defined Exceptions and explain any one with example.
(i.e. Named exception)**

Exception	Raised When...
INVALID_NUMBER	TO_NUMBER function failed in converting string to number.
NO_DATA_FOUND	SELECT ... INTO statement couldn't find data.
ZERO_DIVIDE	Divide by zero error occurred.
TOO_MANY_ROWS	SELECT ... INTO statement found more than one record.
LOGIN_DENIED	Invalid username or password found while logging.
NOT_LOGGED_ON	Statements tried to execute without logging.
INVALID_CURSOR	A cursor is attempted to use which is not open.
PROGRAM_ERROR	PL/SQL found internal problem.
DUP_VAL_ON_INDEX	Duplicate value found in column defined as unique or primary key.
VALUE_ERROR	Error occurred during conversion of data.
OTHERS	Stands for all other exceptions.

Example:

DECLARE

-- declare required variable

no Account.Acc_No%TYPE;

bal Account.Balance%TYPE;

branch Account.B_Name%TYPE;

BEGIN

--read an account number, balance and branch name for new record

no := &no;

```

        bal := &bal;
        branch := &branch;
--insert record into Account table
        INSERT INTO Account VALUES (no, bal, branch);
        --commit and display message confirming insertion
        COMMIT;
        dbms_output.put_line('Record inserted successfully.');
```

EXCEPTION

```

        --handle named exception
        WHEN DUP_VAL_ON_INDEX THEN
        dbms_output.put_line('Duplicate value found for primary
        key.');
```

END;
/

11. Write short note on User Defined Exceptions

- In this case, user has to take care about **declaring an exception, raising it based on some condition** and then **handle** it.

Syntax:

```

DECLARE
        exceptionName      EXCEPTION ;
BEGIN
        --SQL and PL/SQL statement
        IF condition THEN
                RAISE exceptionName
        END IF ;
EXCEPTION
        WHEN exceptionName      THEN
                -- code to Handle Exception
END ;
```

- A user-defined exception can be defined in the declaration section.
- A RAISE clause raises an exception and transfers control of execution from the executable commands section to the exception handling section.
- This exception is handled in the exception handling section.

DECLARE

```
--declare exception and bind it exNull EXCEPTION;
```

```
PRAGMA EXCEPTION_INIT (exNull, -1200);
```

```
--declare exception
```

```
myEx EXCEPTION;
```

```
-- declare required variable
```

```
no Account.Acc_No%TYPE;
```

```
bal Account.Balance%TYPE;
```

```
branch Account.B_Name%TYPE;
```

BEGIN

```
--read an account number, balance and branch name for new record
```

```
no := &no;
```

```
bal := &bal;
```

```
branch := &branch;
```

```
--check balance, if negative, raise 'myEx' exception
```

```
IF bal > 0 THEN
```

```
    RAISE myEx;
```

```
END IF ;
```

```
--insert record into Account table
```

```
INSERT INTO Account VALUES (no, bal, branch);
```

```
--commit and display message confirming insertion
```

```
COMMIT;
```

```
dbms_output.put_line('Record inserted successfully.');
```

EXCEPTION

```
--handle named exception
```

```
WHEN DUP_VAL_ON_INDEX THEN
```

```
    dbms_output.put_line('Duplicate value found for primary  
key.');
```

```
--handle numbered exception
```

```
WHEN exNull THEN
```

```
    dbms_output.put_line('Null value found for primary key.')
```

```
--handle user-defined exception
```

```
WHEN myEx THEN
```

```
    dbms_output.put_line('Balance cannot be negative value.')
```

```
END;
```

```
/
```

12. Write short note on First Normal Form (1NF).

- A relation **R** is in **first normal form** (1NF) if and only if all **domains** contain **atomic** values only.

OR

- A relation **R** is in **first normal form** (1NF) if and only if it does not contain any **composite** or **multi valued attributes** or their combinations.

Example: Customer table

Cid	Name	Address		Contact_No
		Society	City	
C01	Aarav	Nana Bazar, Anand		9879898798,7877855416
C02	Kahaan	C.G. Road, Ahmedabad		9825098254
C03	Pari	M.G. Road, Rajkot		9898787898

- Above relation has four attributes **Cid, Name, Address, Contact_no**. Here **Address** is **composite** attribute which is further divided in to sub attributes as **Society** and **City**. Another attribute **Contact_no** is **multi valued attribute** which can store more than one values. So
- above **relation is not in 1NF**.

Problem:

- Suppose we want to find all customers for some particular city then it is difficult to retrieve. Reason is city name is combined with society name and stored whole as address.

Solution:

- Insert separate attribute for each sub attribute of **composite** attribute.
- Determine maximum allowable values for **multi-valued** attribute.
- Insert separate attribute for **multi valued** attribute and insert only one value on one attribute and other in other attribute.
- So, above table can be created as follows:

Cid	Name	Society	City	Contact_No1	Contact_No2
C01	Aarav	Nana Bazar	Anand	9879898798	7877855416
C02	Kahaan	C.G. Road	Ahmedabad	9825098254	
C03	Pari	M.G. Road	Rajkot	9898787898	

13. Write short note on Second Normal Form (2NF).

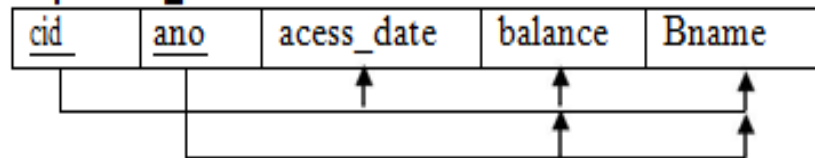
- A relation R is in second normal form (2NF) if and only if it is in 1NF and every non-prime attribute of relation is fully dependent on the primary key.

OR

- A relation R is in second normal form (2NF) if and only if it is in 1NF and no any non-prime attribute is partially dependent on the primary key.

Example:

Depositor_Account:



- Above relation has five attributes **cid**, **ano**, **access_date**, **balance**, **bname** and two FDS
 - FD1: {cid,ano}->{access_date, balance, bname} and
 - FD2: {ano}->{balance, bname}
- We have **cid and ano as primary key**. As per FD2 **balance** and **bname** are only depend on **ano** not **cid**.
- In above table **balance** and **bname** are not fully dependent on primary key but these attributes are partial dependent on primary key.
- So above relation is not in 2NF.

Problem:

- For example in case of joint account **multiple customers** have common **account**. If some account says 'A02' is jointly by two customers says 'C02' and 'C04' then data values for attributes **balance** and **bname** will be duplicated in two different tuples of customers 'C02' and 'C04'.


Solution:

- Decompose relation in such a way that resultant relation does not have any **partial FD**.
- For this purpose remove partial dependent attributes that violets 2NF from relation. Place them in **separate new relation** along with the **prime** attribute on which they are **full dependent**.
- The **primary key** of new relation will be the attribute on which they are fully dependent.
- Keep other attributes same as in that table with same primary key.

- So, above table **Depositor_Account** can be decomposed into **Account** and **Account_Holder** table as per following.

Account:

<u>Ano</u>	balance	bname
------------	---------	-------

**Account_Holder:**

<u>cid</u>	<u>ano</u>	acess_date
------------	------------	------------

