What is a VIEW in Oracle?
An Oracle **VIEW**, in essence, is a **virtual** table that **does not physically exist**. Rather, it is **created** by a **query joining one or more** tables.

Syntax:

```
CREATE VIEW view_name AS
  SELECT columns
  FROM tables
  [WHERE conditions];
```

view_name: The name of the Oracle **VIEW** that you wish to **create**.
WHERE conditions: Optional. The **conditions** that **must** be **met** for the **records** to be **included** in the **VIEW**.

E.g.:

```
CREATE VIEW sup_orders AS
  SELECT suppliers.supplier_id, orders.quantity, orders.price
  FROM suppliers
  INNER JOIN orders
  ON suppliers.supplier_id = orders.supplier_id
  WHERE suppliers.supplier_name = 'Microsoft';
```

This Oracle **CREATE VIEW** example would create a **virtual** table based on the **result set** of the **SELECT** statement.

You can now **query** the Oracle **VIEW** as follows:

```
SELECT *
FROM sup_orders;
```

Update VIEW:
You can modify the definition of an **Oracle VIEW** without **dropping** it by using the Oracle **CREATE** OR **REPLACE** VIEW Statement.

```
CREATE OR REPLACE VIEW view_name AS
  SELECT columns
  FROM table
  WHERE conditions;
```

view_name: The name of the Oracle **VIEW** that you wish to **create** or **replace**.

```
CREATE or REPLACE VIEW sup_orders AS
  SELECT suppliers.supplier_id, orders.quantity, orders.price
  FROM suppliers
  INNER JOIN orders
  ON suppliers.supplier_id = orders.supplier_id
  WHERE suppliers.supplier_name = 'Apple';
```

This Oracle **CREATE** OR **REPLACE VIEW** example would **update** the **definition** of the **Oracle VIEW** called ***sup_orders*** without **dropping** it. If the Oracle VIEW did **not** yet **exist**, the **VIEW** would merely be **created** for the **first time**.

Drop VIEW:
Once an Oracle VIEW has been created, you can **drop** it with the Oracle **DROP VIEW** Statement.

Syntax:

```
DROP VIEW view_name;
```

view_name: The name of the **view** that you wish to **drop**.

E.g.:

```
DROP VIEW sup_orders;
```

This Oracle **DROP VIEW** example would **drop/delete** the Oracle **VIEW** called ***sup_orders***.

When to use the Oracle view:

You can **use views** in **many cases** for **different purposes**. The most common uses of views are as follows:

- Simplifying data retrieval.
- Maintaining logical data independence.
- Implementing data security.

1) Simplifying data retrieval:

Views help **simplify data retrieval significantly**. First, you build a **complex query**, **test** it **carefully**, and **encapsulate** the **query** in a **view**. Then, you can **access** the **data** of the **underlying tables** through the **view** instead of **rewriting** the **whole query** again and again.

The following query **returns sales amount** by the **customer** by **year**:

```
SELECT
    name AS customer,
    SUM( quantity * unit_price ) sales_amount,
    EXTRACT(
        YEAR
    FROM
        order_date
    ) YEAR
FROM
    orders
INNER JOIN order_items
        USING(order_id)
INNER JOIN customers
        USING(customer_id)
WHERE
    status = 'Shipped'
GROUP BY
    name,
    EXTRACT(
        YEAR
    FROM
        order_date
    );
```

Output:

CUSTOMER	SALES_AMOUNT	YEAR
International Paper	613735.06	2015
AutoNation	968134.3	2017
Becton Dickinson	484279.39	2016
Plains GP Holdings	655593.37	2016
Supervalu	394765.27	2017
Centene	417049.24	2017
CenturyLink	711307.09	2016
Abbott Laboratories	697288.63	2016
AutoNation	236531.07	2016
Jabil Circuit	508588.59	2017

This **query** is a **little complex**. Typing it **over** and **over again** is **time-consuming** and **potentially cause** a **mistake**. To **simplify** it, you can **create** a **view** based on this query:

```
CREATE OR REPLACE VIEW customer_sales AS
SELECT
    name AS customer,
    SUM( quantity * unit_price ) sales_amount,
    EXTRACT(
        YEAR
    FROM
        order_date
    ) YEAR
FROM
    orders
INNER JOIN order_items
        USING(order_id)
INNER JOIN customers
        USING(customer_id)
WHERE
    status = 'Shipped'
GROUP BY
    name,
    EXTRACT(
        YEAR
    FROM
        order_date
    );
```

Now, you can **easily retrieve** the **sales** by the **customer** in **2017** with a **simpler query**:

```
SELECT
    customer,
    sales_amount
FROM
    customer_sales
WHERE
    YEAR = 2017
ORDER BY
    sales_amount DESC;
```

CUSTOMER	SALES_AMOUNT
Raytheon	2406081.53
NextEra Energy	1449154.87
Southern	1304990.28
General Mills	1081679.88
AutoNation	968134.3
Emerson Electric	952330.09
Progressive	950118.04
Aflac	698612.98
Goodyear Tire & Rubber	676068.67
AutoZone	645379.54
Jabil Circuit	508588.59

2) Maintaining logical data independence:
You can **expose** the **data** from **underlying tables** to the **external applications** via **views**. Whenever the **structures** of the **base tables change**, you just **need** to **update** the **view**. The **interface between** the **database** and the **external applications remains intact**. The beauty is that you **don't have to change a single line of code** to keep the **external applications up** and **running**.
E.g. Some **reporting systems** may need **only customer sales** data for **composing strategic reports**. Hence, you **can** give the **application owners** the **customer_sales** view.

3) Implementing data security:
Views allow you to **implement** an **additional security layer**. They help you **hide** certain **columns** and **rows** from the **underlying tables** and **expose** only **needed data** to the **appropriate users**.

Oracle **provides** you with **GRANT** and **REVOKE** commands on **views** so that you **can** specify which **actions** a **user** can **perform against** the **view**. **Note** that **in this case**, you **don't grant** any **privileges** on the **underlying tables** because you may **not** want the **user** to **bypass** the **views** and **access** the **base tables directly**.

Q1) Can you update the data in an Oracle VIEW?

A VIEW in Oracle is created by joining one or more tables. When you update record(s) in a VIEW, it updates the records in the underlying tables that make up the View.

So, yes, you can update the data in an Oracle VIEW providing you have the proper privileges to the underlying Oracle tables.

Q2) Does the Oracle View exist if the table is dropped from the database?

Yes, in Oracle, the VIEW continues to exist even after one of the tables (that the Oracle VIEW is based on) is dropped from the database. However, if you try to query the Oracle VIEW after the table has been dropped, you will receive a message indicating that the Oracle VIEW has errors.

If you recreate the table (the table that you had dropped), the Oracle VIEW will again be fine.

What is a Stored Procedure?

- A stored procedure is a PL/SQL block which performs a specific task or a set of tasks.
- A procedure has a name, contains SQL queries and is able to receive parameters and return results.
- A procedure is similar to functions (or methods) in programming languages.

When should I use a Stored Procedure?

When you want to perform a particular task many times, then in place of repeating the queries every time, embed it in a procedure and just call it where ever required.

E.g.

Given the date of birth of an Employee you want to calculate his age and this is required at the time of Employee registration, when applying for insurance of Employee or when transferring him to other departments etc.

Now either you write the logic of age calculation at all the places separately or create a stored procedure with the logic and call it where required.

Benefits of Stored Procedure:

1) Reusability: Create a procedure once and use it any number of times at any number of places. You just need to call it and your task is done.

2) Easy Maintenance: If instead of using a procedure, you repeat the SQL everywhere and if there is a change in logic, then you need to update it at all the places. With stored procedure, the change needs to be done at only one place.

Syntax:

```
CREATE OR REPLACE PROCEDURE <procedure_name>
(<variable_name>IN/OUT/IN OUT <datatype>,
   <variable_name>IN/OUT/IN OUT <datatype>,...) IS/AS
variable/constant declaration;
BEGIN
    -- PL/SQL subprogram body;
EXCEPTION
    -- Exception Handling block ;
END <procedure_name>;
```

Let's understand the above code:

1) procedure_name is for procedure's name and **variable_name** is the variable name for variable used in the stored procedure.

2) CREATE or **REPLACE PROCEDURE** is a keyword used for specifying the name of the procedure to be created.

3) BEGIN, **EXCEPTION** and **END** are keywords used to indicate different sections of the procedure being created.

4) IN/OUT/IN OUT are parameter modes.

5) IN mode refers to **READ ONLY** mode which is used for a variable by which it will **accept** the **value** from the **user**. It is the **default parameter** mode.

6) OUT mode refers to **WRITE ONLY** mode which is used for a variable that will return the value to the user.

7) IN OUT mode refers to **READ AND WRITE** mode which is used for a **variable** that will either **accept** a **value** from the user **OR** it will **return** the **value** to the user.

8) At the end, **<procedure_name>** is **optional** to **write**, you can simply use **END** statement to **end** the **procedure** definition.

E.g.

```
set serveroutput on;

CREATE OR REPLACE PROCEDURE Sum (a IN number, b IN number) IS
c number;
BEGIN
        c := a+b;
        dbms_output.put_line ('Sum of two nos= '|| c);
END Sum;
```

Execute the Procedure:

```
DECLARE
        x number;
        y number;
BEGIN
        x := &x;
        y := &y;
        Sum(x,y);
END;
```

Now that we know how to create stored procedures and how to use them in another PL/SQL code block, it's time to understand how to create Functions in PL/SQL.

```
CREATE OR REPLACE FUNCTION <function_name>
(<variable_name> IN <datatype>,
<variable_name> IN <datatype>,...)
RETURN <datatype> IS/AS
variable/constant declaration;
BEGIN
    -- PL/SQL subprogram body;
EXCEPTION
    -- Exception Handling block ;
END <function_name>;
```

Let's understand the above code:
1) function_name is for defining function's name and **variable_name** is the variable name for variable used in the function.
2) CREATE or REPLACE FUNCTION is a keyword used for specifying the name of the function to be created.
3) IN mode refers to **READ ONLY** mode which is used for a **variable** by which it will **accept** the **value** from the **user**. It is the **default parameter mode**.
4) RETURN is a keyword followed by a **datatype** specifying the **datatype** of a **value** that the **function** will **return**.
Note: Most part of the above code is similar to the one used for defining a stored procedure.

```
set serveroutput on;

CREATE OR REPLACE FUNCTION fn_Sum (a IN number, b IN number) RETURN Number IS
c number;
BEGIN
        c := a+b;
        RETURN c;
END;
```

For **calling** the **function** fn_Sum **following code** will be executed:

```
DECLARE
        no1 number;
        no2 number;
        result number;
BEGIN
        no1 := &no1;
        no2 := &no2;
        result := fn_Sum(no1,no2);
dbms_output.put_line('Sum of two nos='||result);
END;
```

Differences between stored procedure and function in PL/SQL:

Stored Procedure	Function
May or may not returns a value to the calling part of program.	Returns a value to the calling part of the program.
Uses IN, OUT, IN OUT parameter.	Uses only IN parameter.
Returns a value using 'OUT' parameter.	Returns a value using 'RETURN'.
Does not specify the datatype of the value if it is going to return after a calling made to it.	Necessarily specifies the datatype of the value which it is going to return after a calling made to it.
Cannot be called from the function block of code.	Can be called from the procedure block of code.