Date	Dec 5 2024	Unit	

PLSQL

PLSQL compiler for reference - https://onecompiler.com/plsql/

PL/SQL (Procedural Language/Structured Query Language) is an extension of SQL used in **DBMS (Database Management Systems)**, particularly with **Oracle databases**. It combines the data manipulation capabilities of SQL with procedural programming constructs like loops, conditions, and exceptions. Here's a breakdown of its key features and significance:

1. Purpose of PL/SQL:

- PL/SQL enables users to write **procedural code** to interact with a database, unlike standard SQL, which is declarative.
- It allows embedding SQL statements within its block structure, making it more powerful for complex data operations.

2. Key Features:

- **Block Structure:** PL/SQL programs are organized into blocks, which are the basic units of a PL/SQL program.
    - **Anonymous Block:** Temporary and not stored in the database.
    - **Named Block:** Stored procedures, functions, triggers, etc.
- **Procedural Constructs:** Includes variables, loops (`FOR`, `WHILE`), conditionals (`IF`, `ELSE`), and error handling.
- **Exception Handling:** Provides robust mechanisms to handle runtime errors using `EXCEPTION` blocks.

- **Cursors:** Allows row-by-row processing of query results.
- **Triggers:** Automates actions based on events such as `INSERT`, `UPDATE`, or `DELETE`.
- **Stored Procedures & Functions:** Reusable blocks of code stored in the database.

3. Advantages of PL/SQL:

- **Modularity:** Code can be divided into procedures, functions, and packages.
- **Performance:** Reduces the number of calls between the client and the database by processing data in batches.
- **Security:** Business logic can be stored in the database, controlling access through permissions.
- **Portability:** Code written in PL/SQL can run on any Oracle database.

5. Applications of PL/SQL:

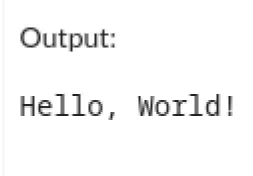
- Automating repetitive tasks like generating reports.
- Implementing complex business logic within the database.
- Creating triggers to enforce constraints or audit changes.
- Building backend components for enterprise applications.

Examples of a PL/SQL

- Displaying a Message

This query demonstrates a basic `BEGIN...END` block in PL/SQL to print a message.

```
DECLARE
  message varchar2(100) := 'Hello, World!';
BEGIN
  dbms_output.put_line(message);
END;
```



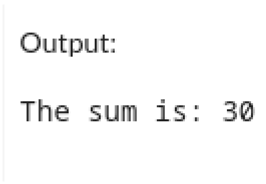
Output:

```
Hello, World!
```

- Adding Two Numbers

This query calculates the sum of two numbers and displays the result.

```
DECLARE
   num1 NUMBER := 10;
   num2 NUMBER := 20;
   sum NUMBER;
BEGIN
   sum := num1 + num2;
   DBMS_OUTPUT.PUT_LINE('The sum is: ' || sum);
END;
```



Output:

```
The sum is: 30
```

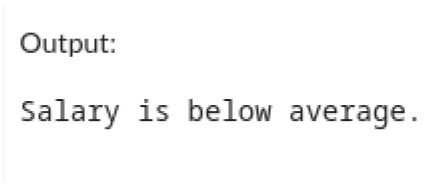
- DECLARE section defines variables.
- The result is stored in the sum variable and printed using DBMS_OUTPUT.PUT_LINE.
- DBMS_OUTPUT.PUT_LINE is used to display messages.
- BEGIN...END defines the PL/SQL block.

- Conditional Logic

This uses `IF...ELSE` to check a condition.

```
DECLARE
   salary NUMBER := 4000;
BEGIN
   IF salary > 5000 THEN
      DBMS_OUTPUT.PUT_LINE('Salary is above average.');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Salary is below average.');
   END IF;
END;
```

- The IF...ELSE statement checks whether the salary is above or below average.
- Outputs the appropriate message.



Output:

```
Salary is below average.
```

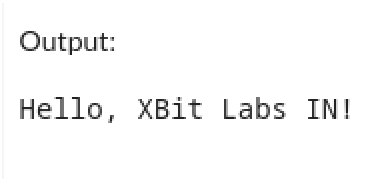
- Create and Call a Procedure

This creates a procedure to display a greeting.

```
CREATE OR REPLACE PROCEDURE greet_user(p_name IN VARCHAR2) AS
BEGIN
   DBMS_OUTPUT.PUT_LINE('Hello, ' || p_name || '!');
END;
/

BEGIN
   greet_user('XBit Labs IN');
END;
```

- CREATE OR REPLACE PROCEDURE defines a reusable procedure.
- The parameter p_name is passed when calling the procedure (greet_user(XBit Labs IN)).



Output:

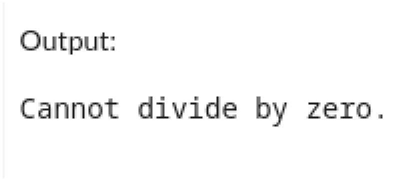
```
Hello, XBit Labs IN!
```

- Handling Exceptions

This example shows how to handle errors gracefully.

```
DECLARE
   num1 NUMBER := 10;
   num2 NUMBER := 0;
   result NUMBER;
BEGIN
   BEGIN
      result := num1 / num2;
      DBMS_OUTPUT.PUT_LINE('Result: ' || result);
   EXCEPTION
      WHEN ZERO_DIVIDE THEN
         DBMS_OUTPUT.PUT_LINE('Cannot divide by zero.');
   END;
END;
```

- The inner BEGIN...END block attempts division.
- If a division by zero occurs, the ZERO_DIVIDE exception is caught, and an error message is displayed.



Output:

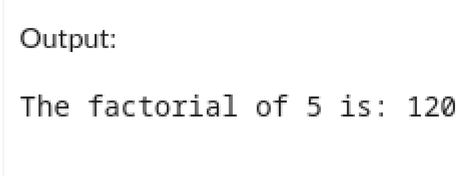
```
Cannot divide by zero.
```

- Calculate the Factorial of a Number

This computes the factorial of a number using a loop.

```
DECLARE
   num NUMBER := 5;
   factorial NUMBER := 1;
BEGIN
   FOR i IN 1..num LOOP
      factorial := factorial * i;
   END LOOP;

   DBMS_OUTPUT.PUT_LINE('The factorial of ' || num || ' is: ' || factorial);
END;
```



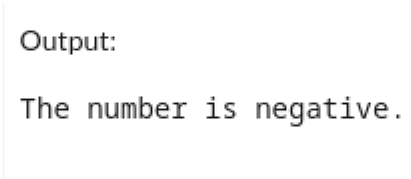
Output:

```
The factorial of 5 is: 120
```

- Use Conditional Logic

This uses `IF...ELSE` statements to classify a number as positive, negative, or zero.

```
DECLARE
   num NUMBER := -7;
BEGIN
   IF num > 0 THEN
      DBMS_OUTPUT.PUT_LINE('The number is positive.');
   ELSIF num < 0 THEN
      DBMS_OUTPUT.PUT_LINE('The number is negative.');
   ELSE
      DBMS_OUTPUT.PUT_LINE('The number is zero.');
   END IF;
END;
```



Output:

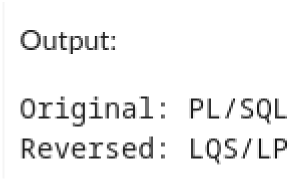
```
The number is negative.
```

- Reverse a String

This demonstrates reversing a string using a loop.

```
DECLARE
   original_string VARCHAR2(50) := 'PL/SQL';
   reversed_string VARCHAR2(50) := '';
BEGIN
   FOR i IN REVERSE 1..LENGTH(original_string) LOOP
      reversed_string := reversed_string || SUBSTR(original_string, i, 1);
   END LOOP;

   DBMS_OUTPUT.PUT_LINE('Original: ' || original_string);
   DBMS_OUTPUT.PUT_LINE('Reversed: ' || reversed_string);
END;
```



Output:

```
Original: PL/SQL
Reversed: LQS/LP
```

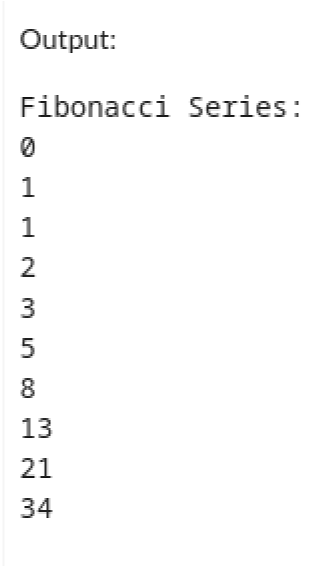
- Fibonacci Series

This generates the first N numbers in the Fibonacci sequence.

```
DECLARE
   n NUMBER := 10;
   a NUMBER := 0;
   b NUMBER := 1;
   next NUMBER;
BEGIN
   DBMS_OUTPUT.PUT_LINE('Fibonacci Series: ');
   DBMS_OUTPUT.PUT_LINE(a);
   DBMS_OUTPUT.PUT_LINE(b);

   FOR i IN 3..n LOOP
      next := a + b;
```

```
        DBMS_OUTPUT.PUT_LINE(next);
        a := b;
        b := next;
    END LOOP;
END;
```



```
Output:

Fibonacci Series:
0
1
1
2
3
5
8
13
21
34
```

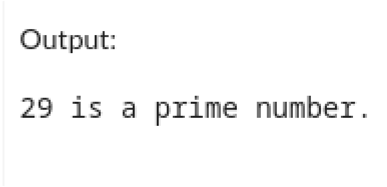
- Check if a Number is Prime

This program checks if a given number is a prime number.

```
DECLARE
    num NUMBER := 29;
    is_prime BOOLEAN := TRUE;
BEGIN
    IF num < 2 THEN
        is_prime := FALSE;
    ELSE
        FOR i IN 2..TRUNC(SQRT(num)) LOOP
            IF MOD(num, i) = 0 THEN
                is_prime := FALSE;
                EXIT;
            END IF;
        END LOOP;
    END IF;

    IF is_prime THEN
        DBMS_OUTPUT.PUT_LINE(num || ' is a prime number.');
```

```
  ELSE
    DBMS_OUTPUT.PUT_LINE(num || ' is not a prime number.');
  END IF;
END;
```



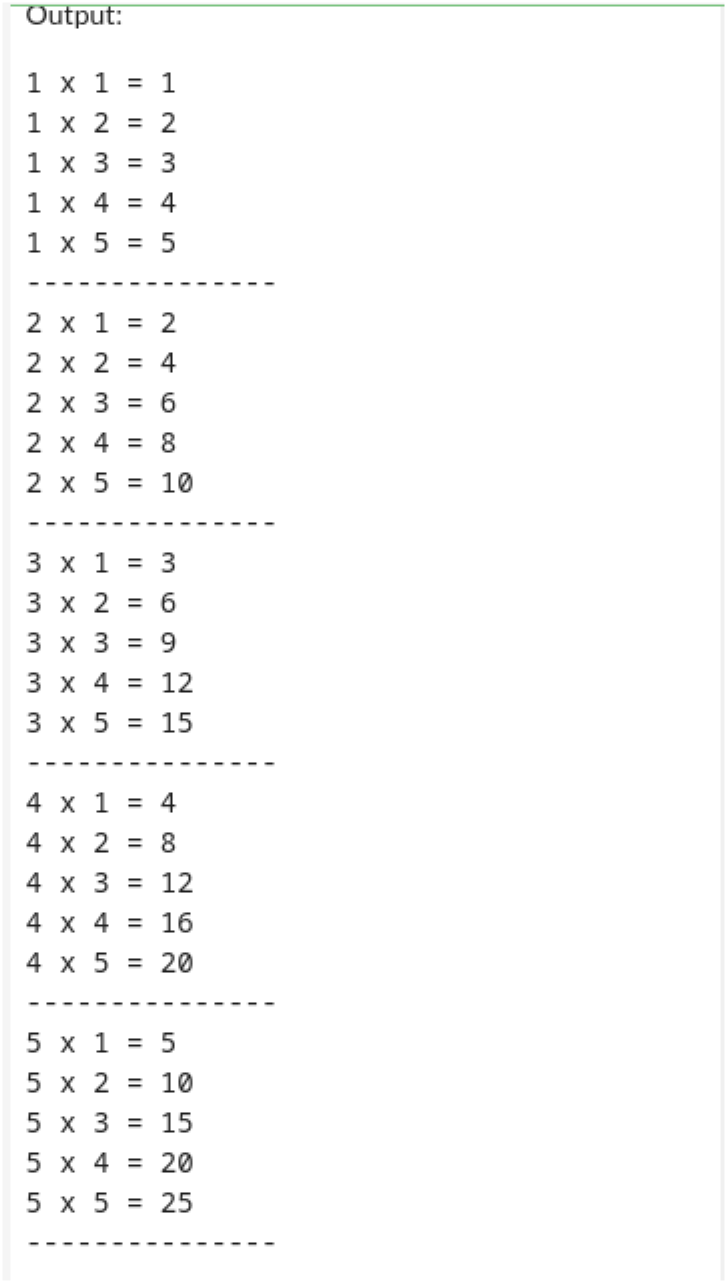
Output:

```
29 is a prime number.
```

- Using Nested Loops

This generates a multiplication table using nested loops.

```
DECLARE
  max_num NUMBER := 5;
BEGIN
  FOR i IN 1..max_num LOOP
    FOR j IN 1..max_num LOOP
      DBMS_OUTPUT.PUT_LINE(i || ' x ' || j || ' = ' || (i * j));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('---------------'); -- Separator
  END LOOP;
END;
```



```
Output:

1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
---------------
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
---------------
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
---------------
4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20
---------------
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
---------------
```

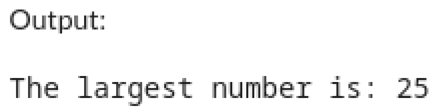
- **Find Maximum of Three Numbers**

This finds the largest of three numbers using simple conditional logic.

```
DECLARE
   num1 NUMBER := 12;
   num2 NUMBER := 25;
   num3 NUMBER := 18;
   max_num NUMBER;
```

```
BEGIN
   IF num1 > num2 AND num1 > num3 THEN
      max_num := num1;
   ELSIF num2 > num3 THEN
      max_num := num2;
   ELSE
      max_num := num3;
   END IF;

   DBMS_OUTPUT.PUT_LINE('The largest number is: ' || max_num);
END;
```



Output:

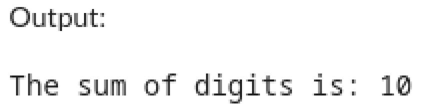
```
The largest number is: 25
```

- Sum of Digits of a Number

This program calculates the sum of digits of a given number.

```
DECLARE
   num NUMBER := 1234;
   sum_of_digits NUMBER := 0;
   remainder NUMBER;
BEGIN
   WHILE num > 0 LOOP
      remainder := MOD(num, 10);
      sum_of_digits := sum_of_digits + remainder;
      num := TRUNC(num / 10);
   END LOOP;

   DBMS_OUTPUT.PUT_LINE('The sum of digits is: ' || sum_of_digits);
END;
```



Output:

```
The sum of digits is: 10
```

The **TRUNC() function** in **PL/SQL** (and SQL) is used to **truncate a value by removing the fractional part** or reducing the precision of the value, depending on its input type. It can be applied to numeric and date values, and its behavior varies accordingly.

LAB Practice

- **Create Views for a Table**

```
-- Assume a table EMPLOYEES (EMP_ID, EMP_NAME, SALARY)

CREATE OR REPLACE VIEW EMPLOYEE_DETAILS AS
SELECT EMP_ID, EMP_NAME
FROM EMPLOYEES;
```

- **Implement Locks for a Table**

```
BEGIN
  -- Lock the EMPLOYEES table in EXCLUSIVE mode
  EXECUTE IMMEDIATE 'LOCK TABLE EMPLOYEES IN EXCLUSIVE MODE';
  -- Perform operations...
  DBMS_OUTPUT.PUT_LINE('Table locked successfully.');
END;
```

- **PL/SQL Procedure with Exception Handling**

```
CREATE OR REPLACE PROCEDURE UpdateSalary(p_emp_id IN NUMBER, p_new_salary IN
NUMBER) AS
BEGIN
   UPDATE EMPLOYEES
   SET SALARY = p_new_salary
   WHERE EMP_ID = p_emp_id;

   IF SQL%ROWCOUNT = 0 THEN
      RAISE_APPLICATION_ERROR(-20001, 'Employee not found');
   END IF;

   DBMS_OUTPUT.PUT_LINE('Salary updated successfully.');
EXCEPTION
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
END;
```

- **PL/SQL Procedure Using Cursors**

```
CREATE OR REPLACE PROCEDURE PrintEmployees IS
   CURSOR emp_cursor IS
      SELECT EMP_ID, EMP_NAME FROM EMPLOYEES;
   emp_record emp_cursor%ROWTYPE;
BEGIN
   OPEN emp_cursor;
   LOOP
      FETCH emp_cursor INTO emp_record;
      EXIT WHEN emp_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('EMP_ID: ' || emp_record.EMP_ID || ', EMP_NAME: ' ||
emp_record.EMP_NAME);
   END LOOP;
   CLOSE emp_cursor;
END;
```

- **PL/SQL Procedure Using Functions**

```
CREATE OR REPLACE FUNCTION CalculateBonus(p_salary IN NUMBER) RETURN
NUMBER IS
BEGIN
   RETURN p_salary * 0.10; -- 10% bonus
END;

CREATE OR REPLACE PROCEDURE AssignBonus(p_emp_id IN NUMBER) AS
   v_salary EMPLOYEES.SALARY%TYPE;
   v_bonus NUMBER;
BEGIN
   SELECT SALARY INTO v_salary FROM EMPLOYEES WHERE EMP_ID = p_emp_id;
   v_bonus := CalculateBonus(v_salary);
   DBMS_OUTPUT.PUT_LINE('Bonus for Employee ' || p_emp_id || ': ' || v_bonus);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Employee not found.');
END;
```

- **PL/SQL Procedure Using Packages**

```
CREATE OR REPLACE PACKAGE EmployeePackage IS
   PROCEDURE GetEmployeeDetails(p_emp_id IN NUMBER);
   PROCEDURE UpdateEmployeeSalary(p_emp_id IN NUMBER, p_salary IN NUMBER);
```

```
END;

CREATE OR REPLACE PACKAGE BODY EmployeePackage IS
  PROCEDURE GetEmployeeDetails(p_emp_id IN NUMBER) IS
    v_name EMPLOYEES.EMP_NAME%TYPE;
    v_salary EMPLOYEES.SALARY%TYPE;
  BEGIN
    SELECT EMP_NAME, SALARY INTO v_name, v_salary FROM EMPLOYEES WHERE
EMP_ID = p_emp_id;
    DBMS_OUTPUT.PUT_LINE('Name: ' || v_name || ', Salary: ' || v_salary);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Employee not found.');
  END;

  PROCEDURE UpdateEmployeeSalary(p_emp_id IN NUMBER, p_salary IN NUMBER) IS
  BEGIN
    UPDATE EMPLOYEES SET SALARY = p_salary WHERE EMP_ID = p_emp_id;
    DBMS_OUTPUT.PUT_LINE('Salary updated for Employee ' || p_emp_id);
  EXCEPTION
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
  END;
END;
```

Package Usage

```
BEGIN
  EmployeePackage.GetEmployeeDetails(101);
  EmployeePackage.UpdateEmployeeSalary(101, 50000);
END;
```

In Oracle, **locks** are mechanisms used to control simultaneous access to data in a database, ensuring **data consistency** and preventing conflicts when multiple users or sessions attempt to access the same resource concurrently.

Types of Locks

Locks in Oracle are broadly categorized based on their **granularity** (e.g., table-level or row-level) and **access control** (e.g., read or write).

1. **Exclusive Lock (X Lock)**:
    - Prevents other users from accessing the table for both reading and writing.
    - Ensures that the locked resource can only be modified by the session that holds the lock.

2. **Share Lock (S Lock)**:
    - Allows other users to read the locked table but prevents them from making changes.
    - Multiple sessions can place a share lock on the same table.
3. **Row Locks**:
    - Locks specific rows in a table rather than the entire table.
    - Prevents other sessions from modifying the locked rows but allows access to other rows.
4. **Implicit Locks**:
    - Automatically applied by Oracle when executing **DML** operations (e.g., `INSERT`, `UPDATE`, `DELETE`).
    - Ensures that changes are not visible to others until committed.
5. **Explicit Locks**:
    - Manually applied by the user using commands like `LOCK TABLE`.

How Table Locks Work

Example 1: Implicit Lock (Automatic)

When a user modifies a table:

```
UPDATE EMPLOYEES
SET SALARY = SALARY + 1000
WHERE EMP_ID = 101;
```

Effect: A row-level lock is placed on the row where `EMP_ID = 101`.
Other users cannot modify the same row until the transaction is committed or rolled back.

Example 2: Explicit Lock (Manual)

Use `LOCK TABLE` to explicitly lock the entire table:

```
LOCK TABLE EMPLOYEES IN EXCLUSIVE MODE;
```

Effect: The entire `EMPLOYEES` table is locked, preventing other sessions from making changes.
Useful for operations where complete table isolation is required (e.g., batch updates).

Modes of Explicit Locking

1. **ROW SHARE (RS)**:
    - Allows concurrent access to the table but prevents other sessions from locking the table in exclusive mode.

```
LOCK TABLE EMPLOYEES IN ROW SHARE MODE;
```

2. **ROW EXCLUSIVE (RX)**:
    - Prevents other sessions from locking the table in share or exclusive mode.
    - Allows DML operations.

```
LOCK TABLE EMPLOYEES IN ROW EXCLUSIVE MODE;
```

3. **SHARE (S)**:
    - Prevents other sessions from modifying the table.
    - Allows other sessions to query the table.

```
LOCK TABLE EMPLOYEES IN SHARE MODE;
```

4. **SHARE ROW EXCLUSIVE (SRX)**:
    - Prevents other sessions from performing DML operations or acquiring share locks.

```
LOCK TABLE EMPLOYEES IN SHARE ROW EXCLUSIVE MODE;
```

5. **EXCLUSIVE (X)**:
    - Completely locks the table. Other sessions cannot query, insert, update, or delete from the table.

```
LOCK TABLE EMPLOYEES IN EXCLUSIVE MODE;
```

Key Points

- **Lock Duration**: Locks persist until the transaction is committed or rolled back.
- **Deadlocks**: Occur when two or more sessions wait for each other’s locks indefinitely. Oracle automatically detects and resolves deadlocks by aborting one of the transactions.
- **Lock Scope**: Locks can apply to rows, blocks, or tables depending on the operation.

When to Use Explicit Locks

1. **Critical Operations**: When a table must be isolated during a specific operation.
2. **Avoiding Deadlocks**: To prevent implicit locks from conflicting in multi-step processes.
3. **Batch Processing**: Ensuring no other operations interfere during a large update or delete.

Views for a table

In SQL, a **view** is a virtual table based on the result of a `SELECT` query. It does not store data physically; instead, it dynamically retrieves data from the underlying tables whenever accessed.

Why Use Views?

1. **Simplification**: Create a simplified interface for complex queries.
2. **Security**: Restrict access to specific columns or rows of a table.
3. **Reusability**: Avoid rewriting complex queries repeatedly.
4. **Data Abstraction**: Provide users with a logical representation of the data.

Creating a view

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Examples

Example 1: Basic View

Suppose you have a table `EMPLOYEES` with the following columns:

- `EMP_ID`
- `EMP_NAME`
- `SALARY`
- `DEPARTMENT`

To create a view showing only employee names and salaries:

```
CREATE OR REPLACE VIEW EMPLOYEE_SALARY AS
SELECT EMP_ID, EMP_NAME, SALARY
FROM EMPLOYEES;
```

Usage

```
SELECT * FROM EMPLOYEE_SALARY;
```

Example 2: View with Conditions

Create a view to show employees from the "IT" department:

```
CREATE OR REPLACE VIEW IT_EMPLOYEES AS
SELECT EMP_ID, EMP_NAME, SALARY
FROM EMPLOYEES
WHERE DEPARTMENT = 'IT';
```

usage

```
SELECT * FROM IT_EMPLOYEES;
```

Example 3: Aggregated Data in a View

Create a view to display average salary per department:

```
CREATE OR REPLACE VIEW AVG_SALARY_PER_DEPT AS
SELECT DEPARTMENT, AVG(SALARY) AS AVG_SALARY
FROM EMPLOYEES
GROUP BY DEPARTMENT;
```

usage

```
SELECT * FROM AVG_SALARY_PER_DEPT;
```

Example 4: View with Joins

Suppose you have another table `DEPARTMENTS`:

- `DEPT_ID`
- `DEPT_NAME`

To create a view that joins `EMPLOYEES` and `DEPARTMENTS`:

```
CREATE OR REPLACE VIEW EMP_DEPT_VIEW AS
SELECT E.EMP_ID, E.EMP_NAME, E.SALARY, D.DEPT_NAME
FROM EMPLOYEES E
JOIN DEPARTMENTS D ON E.DEPARTMENT = D.DEPT_ID;
```

Usage

```
SELECT * FROM EMP_DEPT_VIEW;
```

Example 5: Updateable View (Optional)

Some views can be updated if they meet specific criteria (e.g., based on a single table, no group functions).

```
CREATE OR REPLACE VIEW EMPLOYEE_BASIC_INFO AS
SELECT EMP_ID, EMP_NAME
FROM EMPLOYEES;
```

update the table through this view:

```
UPDATE EMPLOYEE_BASIC_INFO
SET EMP_NAME = 'John Doe'
WHERE EMP_ID = 101;
```

Key Points

1. **Materialized Views**: Unlike regular views, materialized views store data physically and are used for performance optimization.
2. **Restrictions**: Views based on joins, group functions, or aggregations are typically read-only.
3. **Modification**: Use `CREATE OR REPLACE` to modify an existing view.
4. **Dropping a View**: Use the `DROP VIEW` command to delete a view.

   ```
   DROP VIEW view_name;
   ```

END