SQL Function Fix

Sure! Here's a complete set of **user-defined functions and procedures** with full **PL/SQL code**, ready to run in Oracle SQL. I've also added **more useful functions** to enrich the set.

🔹 1. Function: `max_val(x, y)` – Return Max Value

sql

```sql
CREATE OR REPLACE FUNCTION max_val(x IN NUMBER, y IN NUMBER)
RETURN NUMBER IS
  z NUMBER;
BEGIN
  IF x > y THEN
    z := x;
  ELSE
    z := y;
  END IF;
  RETURN z;
END;
/
```

🔹 2. Function: `calculate_avg_salary(dep_id)` – Average Salary by Department

sql

```sql
CREATE OR REPLACE FUNCTION calculate_avg_salary(dep IN NUMBER)
RETURN NUMBER IS
  avg_salary NUMBER;
BEGIN
  SELECT AVG(salary) INTO avg_salary
  FROM employees
  WHERE dep_id = dep;
  RETURN avg_salary;
```

```
END;
/
```

🔹 3. Function: `get_emp_count(dep_id)` – Count of Employees in Department

```sql
CREATE OR REPLACE FUNCTION get_emp_count(dep IN NUMBER)
RETURN NUMBER IS
  emp_count NUMBER;
BEGIN
  SELECT COUNT(*) INTO emp_count
  FROM employees
  WHERE dep_id = dep;
  RETURN emp_count;
END;
/
```

🔹 4. Function: `is_high_salary(salary)` – Boolean Check on Salary

```sql
CREATE OR REPLACE FUNCTION is_high_salary(salary IN NUMBER)
RETURN VARCHAR2 IS
BEGIN
  IF salary > 75000 THEN
    RETURN 'YES';
  ELSE
    RETURN 'NO';
  END IF;
END;
/
```

🔹 5. Procedure: `max(x, y)` – Print Sum (Basic Example)

```sql
CREATE OR REPLACE PROCEDURE max(x IN NUMBER, y IN NUMBER)
IS
  sum_result NUMBER;
BEGIN
  sum_result := x + y;
  DBMS_OUTPUT.PUT_LINE('Sum: ' || sum_result);
END;
/
```

🔹 6. Procedure: `get_bonus(emp_salary, bonus OUT)`

```sql
CREATE OR REPLACE PROCEDURE get_bonus(
  emp_salary IN NUMBER,
  bonus OUT NUMBER
) IS
BEGIN
  IF emp_salary >= 50000 THEN
    bonus := emp_salary * 0.1;
  ELSE
    bonus := emp_salary * 0.2;
  END IF;
END;
/
```

🔹 7. Procedure: `register_employee(salary, dep_id)` – Insert Employee

sql

```
CREATE OR REPLACE PROCEDURE register_employee(
  salary IN NUMBER,
  dep_id IN NUMBER
) IS
BEGIN
  INSERT INTO employees(salary, dep_id)
  VALUES (salary, dep_id);
  COMMIT;
  DBMS_OUTPUT.PUT_LINE('Employee registered.');
END;
/
```

🔹 8. Procedure: `list_employees()` – List All Employees with Cursor

sql

```sql
CREATE OR REPLACE PROCEDURE list_employees IS
  CURSOR emp_cursor IS
    SELECT salary, dep_id FROM employees;
  rec emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO rec;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Salary: ' || rec.salary || ', Dept: ' || rec.dep_id);
  END LOOP;
  CLOSE emp_cursor;
END;
/
```

🔹 9. Procedure: `increase_salary_by_dept(p_dept_id, p_percent)` – Bulk Update

```sql
CREATE OR REPLACE PROCEDURE increase_salary_by_dept(
  p_dept_id IN NUMBER,
  p_percent IN NUMBER
) IS
BEGIN
  UPDATE employees
  SET salary = salary + (salary * p_percent / 100)
  WHERE dep_id = p_dept_id;

  DBMS_OUTPUT.PUT_LINE('Salaries increased by ' || p_percent || '% for dept: ' ||
p_dept_id);
  COMMIT;
END;
/
```

🔹 10. Procedure: `safe_divide(a, b, result OUT)` – With Exception Handling

```sql
CREATE OR REPLACE PROCEDURE safe_divide(
  a IN NUMBER,
  b IN NUMBER,
  result OUT NUMBER
) IS
BEGIN
  IF b = 0 THEN
    RAISE_APPLICATION_ERROR(-20001, 'Cannot divide by zero');
  ELSE
    result := a / b;
  END IF;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
/
```

✅ Setup Tables (if not already created)

sql

```sql
CREATE TABLE department (
  id NUMBER PRIMARY KEY
);

CREATE TABLE employees (
  salary NUMBER,
  dep_id NUMBER,
  FOREIGN KEY(dep_id) REFERENCES department(id)
);
```

Sample Data:

sql

```sql
INSERT INTO department VALUES (1);
INSERT INTO department VALUES (2);

INSERT INTO employees VALUES (60000, 1);
INSERT INTO employees VALUES (80000, 1);
INSERT INTO employees VALUES (40000, 2);
COMMIT;
```

Would you like a single `.sql` file containing all of this? Or an example of using these in a report query?

Triggers in SQL are **special stored procedures** that automatically run when certain events occur on a table or view, such as `INSERT`, `UPDATE`, or `DELETE`.

✅ Basic Syntax of a Trigger in Oracle SQL:

```sql
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF}
{INSERT | UPDATE | DELETE}
ON table_name
[FOR EACH ROW]
BEGIN
  -- trigger logic here
END;
/
```

🔹 1. Trigger to Audit Employee Inserts

```sql
CREATE TABLE emp_audit (
  emp_id NUMBER,
  salary NUMBER,
  action_time DATE
);

CREATE OR REPLACE TRIGGER trg_audit_employee
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
  INSERT INTO emp_audit(emp_id, salary, action_time)
  VALUES (:NEW.dep_id, :NEW.salary, SYSDATE);
END;
/
```

This trigger logs each insert into `employees`.

🔹 2. Trigger to Prevent Negative Salaries

sql

```sql
CREATE OR REPLACE TRIGGER trg_no_negative_salary
BEFORE INSERT OR UPDATE ON employees
FOR EACH ROW
BEGIN
  IF :NEW.salary < 0 THEN
    RAISE_APPLICATION_ERROR(-20002, 'Salary cannot be negative');
  END IF;
END;
/
```

This will block inserts or updates with salary < 0.

🔹 3. Trigger on DELETE – Archive Data

sql

```sql
CREATE TABLE emp_deleted (
  salary NUMBER,
  dep_id NUMBER,
  deleted_at DATE
);

CREATE OR REPLACE TRIGGER trg_archive_deleted
BEFORE DELETE ON employees
FOR EACH ROW
BEGIN
  INSERT INTO emp_deleted(salary, dep_id, deleted_at)
  VALUES (:OLD.salary, :OLD.dep_id, SYSDATE);
END;
/
```

This trigger archives deleted employee records.

🔹 4. Trigger to Auto Capitalize Department IDs (String Example)

```sql
CREATE OR REPLACE TRIGGER trg_uppercase_dept
BEFORE INSERT OR UPDATE ON department
FOR EACH ROW
BEGIN
  :NEW.id := UPPER(:NEW.id);
END;
/
```

Only applicable if `id` were a `VARCHAR2` instead of `NUMBER`.

🔎 Viewing All Triggers

To list all user-defined triggers:

```sql
SELECT trigger_name, table_name, triggering_event, status
FROM user_triggers;
```

Would you like an **AFTER UPDATE trigger**, or a **compound trigger example**?

Certainly! Triggers can contain multiple queries, including SELECT, INSERT, UPDATE, and DELETE operations. Here's an example of a more complex **AFTER INSERT trigger** that includes multiple queries to interact with multiple tables.

🔹 Example: Trigger to Update Employee Performance Based on Salary

Scenario:

We want to track employee performance in a `performance` table, based on the salary of the employee. If an employee's salary is above a certain threshold, we want to insert a performance record for them. Additionally, we want to update the department's average salary after an employee's salary has been inserted.

Steps:

1. After a new employee is inserted into the `employees` table:
    - If the salary is greater than `50000`, insert a record into the `performance` table.
    - Update the department's average salary in the `department` table.

SQL Code for the Trigger:

```sql
-- Create Performance Table
CREATE TABLE performance (
  emp_id NUMBER,
  performance_status VARCHAR2(50),
  performance_date DATE
);

-- Create Department Table
CREATE TABLE department (
  id NUMBER PRIMARY KEY,
  avg_salary NUMBER
);

-- Create Employees Table
CREATE TABLE employees (
  emp_id NUMBER PRIMARY KEY,
  salary NUMBER,
  dep_id NUMBER,
  FOREIGN KEY(dep_id) REFERENCES department(id)
);

-- Trigger to Update Performance and Department Average Salary
```

```
CREATE OR REPLACE TRIGGER trg_employee_performance
AFTER INSERT ON employees
FOR EACH ROW
DECLARE
  -- Declare local variables
  v_avg_salary NUMBER;
BEGIN
  -- 1. Insert performance record based on salary
  IF :NEW.salary > 50000 THEN
    INSERT INTO performance(emp_id, performance_status, performance_date)
    VALUES (:NEW.emp_id, 'High Performer', SYSDATE);
  ELSE
    INSERT INTO performance(emp_id, performance_status, performance_date)
    VALUES (:NEW.emp_id, 'Needs Improvement', SYSDATE);
  END IF;

  -- 2. Update department's average salary
  SELECT AVG(salary) INTO v_avg_salary
  FROM employees
  WHERE dep_id = :NEW.dep_id;

  UPDATE department
  SET avg_salary = v_avg_salary
  WHERE id = :NEW.dep_id;

  -- Optional: You could raise an error or log here if necessary
  DBMS_OUTPUT.PUT_LINE('Performance updated and department avg salary modified.');

END;
/
```

Breakdown of the Trigger:

1. `AFTER INSERT ON employees`:
   - This trigger fires after a new record is inserted into the `employees` table.
2. **Insert Performance Record**:
   - We check if the employee's salary is greater than `50,000`. If yes, they are considered a **High Performer**; otherwise, they are labeled as **Needs Improvement**.

  - The performance record is inserted into the `performance` table.

3. **Update Department's Average Salary**:
   - The trigger calculates the new **average salary** for the department using a `SELECT` query.
   - Then, it updates the corresponding department's average salary in the `department` table.

Example Data:

sql

```sql
-- Insert some departments
INSERT INTO department (id) VALUES (1);
INSERT INTO department (id) VALUES (2);

-- Insert employees into the employees table
INSERT INTO employees (emp_id, salary, dep_id) VALUES (101, 60000, 1);
INSERT INTO employees (emp_id, salary, dep_id) VALUES (102, 45000, 1);
INSERT INTO employees (emp_id, salary, dep_id) VALUES (103, 70000, 2);
```

Expected Output:

- When you insert a new employee, the trigger will:
  - Insert a **performance record** based on the salary.
  - Calculate the **average salary** for the department and update it in the `department` table.
- After inserting an employee with a salary of `60000` into department `1`, the `performance` and `department` tables will be updated.

Sample Query to Check Results:

sql

```sql
-- Check the performance records
SELECT * FROM performance;

-- Check the department average salary
SELECT * FROM department;
```

🔎 Summary:

This trigger uses multiple queries to achieve the following:

1. Insert a record in one table (`performance`).
2. Update another table (`department`).
3. Perform conditional logic based on the inserted data (`salary > 50000`).

Would you like to see a **compound trigger** (which allows multiple logic in a more structured way), or any specific examples like **triggers on UPDATE** or **DELETE**?