### 1. ****What is the Purpose of a Stored Procedure?****

Stored procedures are used to **perform actions (primarily DML and transaction** )on a database rather than directly returning data. They are primarily useful for:

* **Encapsulating Business Logic**: Combine multiple SQL statements, conditional logic, or loops in one reusable block.
* **Executing Non-Query Tasks**: For example:
  + Inserting/updating/deleting records.
  + Managing transactions (e.g., committing/rolling back changes).
  + Running batch operations or complex workflows.

Stored procedures **don't return result sets** directly to the caller (though you can log or raise notices). For queries returning rows of data, **functions** are more appropriate.

### 2. ****Difference Between Stored Procedures and Functions****

| ****Feature**** | ****Stored Procedure**** | ****Function**** |
| --- | --- | --- |
| **Purpose** | Execute actions/tasks (e.g., update, delete). | Return a result or perform calculations. |
| **Return Values** | Cannot directly return a result set. | Can return scalar values, rows, or tables. |
| **Calling Syntax** | Use CALL procedure\_name();. | Use SELECT function\_name();. |
| **Usable in Queries** | Cannot be used inside a query. | Can be called inside SQL queries. |
| **Transaction Control** | Can start, commit, or rollback transactions. | Cannot control transactions (runs in one). |

### Example Use Cases:

* **Procedure**: Update the status of orders based on some business logic.
* **Function**: Return the names of all employees with a salary above 50,000.

### 3. ****How to See Existing Stored Procedures****

To view the stored procedures in your database:

#### ****In DBeaver:****

1. Expand the **database name** in the Database Navigator.
2. Navigate to **Schemas → your schema (e.g.,**public**) → Procedures**.
3. You'll see all stored procedures listed here.

#### ****In SQL Query:****

You can use the system catalog pg\_proc to query stored procedures:

sql

Copy code

SELECT proname AS procedure\_name, prosrc AS source\_code

FROM pg\_proc

WHERE prokind = 'p'; -- 'p' denotes procedures

This will list all stored procedures along with their source code.

### 4. ****How to Call a Stored Procedure****

In PostgreSQL, stored procedures are called using the CALL statement. Example:

sql

Copy code

CALL high\_salaries();

### 5. ****Why Use Stored Procedures?****

Stored procedures are valuable when you need to:

* Perform **batch updates/inserts** without needing to return data.
* Run **complex workflows** like loops, conditional logic, or dynamic SQL.
* Encapsulate repetitive logic to avoid duplicating it in multiple queries.

If you need to **return data**, switch to a function.

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

**Part-2 EXAMPLES**

f you need to return data directly (like the SELECT query in your example), a **function** is preferable. However, stored procedures have distinct use cases. Below are some practical examples where **stored procedures** shine.

### 1. ****Batch Updates or Inserts****

Stored procedures are great for updating or inserting multiple records based on conditions or loops.

#### Example: Updating Employee Salaries in Bulk

sql

Copy code

CREATE OR REPLACE PROCEDURE update\_salaries(department\_name TEXT, increment NUMERIC)

AS $$

BEGIN

UPDATE employees

SET salary = salary + increment

WHERE department = department\_name;

RAISE NOTICE 'Salaries updated for department: %', department\_name;

END;

$$ LANGUAGE plpgsql;

-- Call the procedure

CALL update\_salaries('HR', 5000);

### 2. ****Transaction Management****

Stored procedures can handle transactions (e.g., commit/rollback), which **functions cannot do**.

#### Example: Transfer Money Between Accounts

sql

Copy code

CREATE OR REPLACE PROCEDURE transfer\_funds(from\_account INT, to\_account INT, amount NUMERIC)

AS $$

BEGIN

BEGIN

UPDATE accounts

SET balance = balance - amount

WHERE account\_id = from\_account;

UPDATE accounts

SET balance = balance + amount

WHERE account\_id = to\_account;

-- Ensure no negative balance

IF (SELECT balance FROM accounts WHERE account\_id = from\_account) < 0 THEN

ROLLBACK;

RAISE EXCEPTION 'Insufficient funds in account %', from\_account;

ELSE

COMMIT;

END IF;

END;

END;

$$ LANGUAGE plpgsql;

-- Call the procedure

CALL transfer\_funds(101, 102, 500);

### 3. ****Dynamic SQL Execution****

Stored procedures can execute dynamic SQL queries using EXECUTE.

#### Example: Deleting Records Dynamically

sql

Copy code

CREATE OR REPLACE PROCEDURE delete\_records(table\_name TEXT, condition TEXT)

AS $$

BEGIN

EXECUTE format('DELETE FROM %I WHERE %s', table\_name, condition);

RAISE NOTICE 'Records deleted from table: %', table\_name;

END;

$$ LANGUAGE plpgsql;

-- Call the procedure

CALL delete\_records('employees', 'salary < 30000');

### 4. ****Complex Workflows****

Stored procedures are helpful when you need to combine multiple steps with logic in a single operation.

#### Example: Monthly Payroll Processing

sql

Copy code

CREATE OR REPLACE PROCEDURE process\_payroll()

AS $$

BEGIN

-- 1. Calculate bonuses

UPDATE employees

SET salary = salary + bonus

WHERE bonus IS NOT NULL;

-- 2. Log payroll processing

INSERT INTO payroll\_logs (log\_time, status)

VALUES (NOW(), 'Payroll Processed');

RAISE NOTICE 'Payroll processing completed.';

END;

$$ LANGUAGE plpgsql;

-- Call the procedure

CALL process\_payroll();

### 5. ****Audit or Logging****

Stored procedures can be used to record changes or log user activities.

#### Example: Logging Deletions

sql

Copy code

CREATE OR REPLACE PROCEDURE log\_deletion(table\_name TEXT, deleted\_by TEXT)

AS $$

BEGIN

INSERT INTO audit\_logs (table\_name, deleted\_at, deleted\_by)

VALUES (table\_name, NOW(), deleted\_by);

RAISE NOTICE 'Deletion logged for table: %', table\_name;

END;

$$ LANGUAGE plpgsql;

-- Call the procedure

CALL log\_deletion('employees', 'admin\_user');

### When to Use a Stored Procedure:

* **Complex transactions**: When you need multiple steps or rollback capabilities.
* **Dynamic operations**: Executing SQL dynamically based on input.
* **Bulk changes**: Applying the same logic to multiple records (e.g., batch updates).
* **Audit/logging**: Keeping track of changes in the database.

**Part-3 Definition of AS**

**usually we are naming something we should use AS before table like employee\_name AS e. But we are using AS at the Begin what it means?**

###### ChatGPT said:

When you use AS in SQL, its purpose depends on the context in which it's used:

### 1. ****Alias for Columns or Tables:****

* + This is the most common usage of AS, where it is used to assign an alias (shortened or descriptive name) to a column or table.
  + Example:

sql

Copy code

SELECT employee\_name AS e\_name, department AS dept

FROM employees;

Here, AS e\_name gives the alias e\_name to the employee\_name column.

* + For tables:

sql

Copy code

SELECT e.name, e.salary

FROM employees AS e;

Here, AS e gives the alias e to the employees table, which can then be used to reference it.

### 2. ****In PL/pgSQL for Defining Code Blocks:****

* + In procedural SQL (like in PostgreSQL or PL/SQL), AS is used differently.
  + When you define a procedure or a function, AS introduces the code block or body of the procedure/function.
  + Example:

sql

Copy code

CREATE OR REPLACE FUNCTION high\_salaries()

RETURNS void

LANGUAGE plpgsql

AS $$

BEGIN

-- Body of the function

END;

$$;

Here, AS $$ ... $$ encloses the body of the function or procedure.

### Why AS at the BEGIN Level in Your Code?

In your screenshot, AS is being used **incorrectly**. The syntax:

sql

Copy code

CREATE PROCEDURE high\_salaries

AS

BEGIN

...

END;

is invalid in PostgreSQL because AS cannot appear in that position. Instead:

* + Use LANGUAGE plpgsql AS $$ ... $$ to properly define the body.
  + The incorrect use of AS here seems like a confusion with syntax from Oracle PL/SQL or SQL Server.

### Key Difference:

* + AS **before tables/columns**: Used for aliasing or renaming.
  + AS **in procedures/functions**: Marks the body in procedural SQL, not used standalone.

##### You said:

SO AS in storedprocdure and function are different from as alias in stored procedure AS denotes that they are the start of the code block right?

###### ChatGPT said:

Exactly! 🎯

In **stored procedures and functions**, the AS keyword signals the **start of the code block** or **definition of the procedure/function's body**. It differs from its use as an alias in standard SQL queries.

### Differences:

#### 1. AS****in Stored Procedures/Functions****:

* + Used to introduce the **code block**.
  + Paired with LANGUAGE and often enclosed in $$ delimiters (in PostgreSQL).
  + Example:

sql

Copy code

CREATE FUNCTION high\_salaries()

RETURNS void

LANGUAGE plpgsql

AS $$

BEGIN

-- Code block starts here

RAISE NOTICE 'Hello, world!';

END;

$$;

* + Here, AS $$ ... $$ defines where the function's logic begins and ends.

#### 2. AS****for Aliases in SQL Queries****:

* + Used to **rename a column or table** for better readability or brevity in queries.
  + Example:

sql

Copy code

SELECT employee\_name AS e\_name, department AS dept

FROM employees AS e;

* + This has nothing to do with procedures or functions; it's just for simplifying query writing.

### In Summary:

* + **In stored procedures/functions:** AS introduces the block of executable code and marks where the function or procedure logic is written.
  + **In queries:** AS is purely for creating aliases for columns or tables, improving clarity and brevity.

So, the role of AS is **context-dependent**:

* + In procedural SQL: **Start of a code block**.
  + In SQL queries: **Alias/renaming**.

**PART-3 ERROR ANALYSIS**

The error occurs because the PostgreSQL stored procedure update\_salaries is expecting arguments of specific types, but the types of arguments you provided do not match.

### ****Error Analysis****

text

Copy code

ERROR [42883]: procedure update\_salaries(unknown, integer) does not exist

Hint: No procedure matches the given name and argument types.

#### Root Cause:

1. **Type Mismatch in Parameters**:
   * The update\_salaries procedure is defined to accept parameters as:
     + department\_name → TEXT
     + increment → NUMERIC
   * However, when you call it:

sql

Copy code

CALL update\_salaries('DataEngineer', 7000);

* + - 'DataEngineer' is treated as a TEXT (correct).
    - 7000 is treated as an INTEGER by PostgreSQL, while the procedure expects NUMERIC.

### ****Solution****

To fix the type mismatch, you need to explicitly cast 7000 to NUMERIC. Update the CALL statement as follows:

sql

Copy code

CALL update\_salaries('DataEngineer', 7000::NUMERIC);

### ****Why This Happens****

PostgreSQL tries to match the parameter types of the stored procedure exactly. If there's no exact match, it raises an error. In this case:

* 7000 is treated as an INTEGER, but the procedure expects a NUMERIC.
* Since no procedure matches the exact signature (TEXT, INTEGER), PostgreSQL reports that the procedure does not exist.

### How to Avoid Similar Issues

1. **Be Explicit with Types**:
   * Always ensure the data types in the CALL statement match the procedure's parameter types.
   * For example:

sql

Copy code

CALL procedure\_name('string'::TEXT, 123::NUMERIC);

1. **Use Compatible Types**:
   * If you're defining a procedure and anticipate integer or decimal inputs, consider using a type like DOUBLE PRECISION or NUMERIC to support both.
2. **Debugging Tools**:
   * You can check the parameter types of a stored procedure by querying the system catalog:

sql

Copy code

SELECT proname, proargtypes::regtype[] AS argument\_types

FROM pg\_proc

WHERE proname = 'update\_salaries';