# MySQL Procedures vs PostgreSQL Functions for SELECT ALL

## Introduction

When working with databases, retrieving all rows from a table using SELECT \* is a common operation. Both MySQL and PostgreSQL provide ways to encapsulate this logic using stored procedures (MySQL) or functions (PostgreSQL). However, their implementations and use cases differ significantly.

This document explores the differences between MySQL Procedures and PostgreSQL Functions, their performance implications, use cases, and best practices for executing SELECT ALL operations.

## 1. MySQL Procedures

### Overview

Stored procedures in MySQL allow you to encapsulate procedural logic into a reusable block of SQL statements. Procedures are well-suited for tasks that involve data manipulation (e.g., INSERT, UPDATE, DELETE) or batch processing. While procedures can execute SELECT statements, they are not primarily designed for returning result sets to SQL queries.

### Key Characteristics

* **Output Handling**:
  + Procedures do not directly return result sets as output parameters.
  + Instead, results are sent implicitly to the client or application layer.
* **Use Case**:
  + Ideal for complex operations, like transactional logic or operations involving multiple queries.
* **Execution**:
  + MySQL procedures are compiled once and can be executed multiple times without recompilation, making them efficient for repeated executions.

### Example of MySQL Procedure for SELECT ALL

DELIMITER $$  
  
CREATE PROCEDURE GetAllEmployees()  
BEGIN  
 SELECT \* FROM employees;  
END$$  
  
DELIMITER ;

### Usage

To call the procedure:

CALL GetAllEmployees();

The result is returned to the client but cannot be embedded directly in a SQL query.

## 2. PostgreSQL Functions

### Overview

Functions in PostgreSQL are highly versatile and designed to return values or result sets. Unlike MySQL procedures, PostgreSQL functions can be embedded directly in SQL queries, making them ideal for SELECT operations. Functions are compiled and optimized with execution plans for better performance on repeated execution.

### Key Characteristics

* **Return Value**:
  + Functions can return scalar values, a single row, or an entire result set (e.g., a table).
  + The RETURNS TABLE clause allows functions to directly return rows to the caller.
* **Integration with Queries**:
  + Functions can be used in SQL queries, such as SELECT, WHERE, or JOIN clauses.
* **Performance Optimization**:
  + Functions can be declared as IMMUTABLE, STABLE, or VOLATILE to optimize execution plans:
    - **IMMUTABLE**: No side effects; always returns the same result for the same input.
    - **STABLE**: Result does not change within a single query execution.
    - **VOLATILE**: Function’s result may change with every execution.

### Example of PostgreSQL Function for SELECT ALL

CREATE OR REPLACE FUNCTION get\_all\_employees()  
RETURNS TABLE (id INT, name TEXT, department TEXT) AS $$  
BEGIN  
 RETURN QUERY SELECT id, name, department FROM employees;  
END;  
$$ LANGUAGE plpgsql STABLE;

### Usage

To call the function:

SELECT \* FROM get\_all\_employees();

The result is returned as a table and can be integrated into other SQL queries.

### Function Categories

* **IMMUTABLE**:
  + These functions always return the same result when given the same input arguments.
  + They do not modify the database or depend on any table data.
  + Ideal for mathematical calculations or deterministic operations.
  + Example:
  + CREATE OR REPLACE FUNCTION add\_numbers(a INT, b INT)  
    RETURNS INT AS $$  
    BEGIN  
     RETURN a + b;  
    END;  
    $$ LANGUAGE plpgsql IMMUTABLE;
* **STABLE**:
  + These functions return consistent results within a single table scan but might produce different results in different SQL statements.
  + They can read table data but do not modify it.
  + Suitable for functions that query data without side effects.
  + Example:
  + CREATE OR REPLACE FUNCTION get\_employee\_count(dept\_name TEXT)  
    RETURNS INT AS $$  
    BEGIN  
     RETURN (SELECT COUNT(\*) FROM employees WHERE department = dept\_name);  
    END;  
    $$ LANGUAGE plpgsql STABLE;
* **VOLATILE**:
  + These functions can return different results even within a single table scan.
  + They can modify the database and perform operations that depend on the current state.
  + Use when the function involves non-deterministic operations or side effects.
  + Example:
  + CREATE OR REPLACE FUNCTION get\_current\_timestamp()  
    RETURNS TIMESTAMP AS $$  
    BEGIN  
     RETURN CURRENT\_TIMESTAMP;  
    END;  
    $$ LANGUAGE plpgsql VOLATILE;

## 3. Comparison: MySQL Procedure vs PostgreSQL Function

| Feature | MySQL Procedure | PostgreSQL Function |
| --- | --- | --- |
| Primary Purpose | Encapsulation of procedural logic for batch processing or transactions. | Encapsulation of logic for returning values or result sets. |
| Return Value | Does not return a result set directly. | Can return a result set directly (table/rows). |
| Usage in SQL Queries | Cannot be used inside SELECT or queries. | Can be used inside SELECT, WHERE, etc. |
| Performance Optimization | Compiled once, but less optimized for result retrieval. | Cached execution plans improve performance for repeated queries. |
| Transaction Control | Can include explicit COMMIT and ROLLBACK. | Cannot include transaction control (COMMIT, ROLLBACK). |
| Complex Logic | Suitable for complex, multi-step operations. | Suitable for data retrieval and computation. |
| Execution Plan Reuse | Recompiled for each session. | Optimized query execution plan is reused. |
| Portability | Supported in MySQL only. | Supported in PostgreSQL and widely portable. |

## 4. When to Use MySQL Procedures

MySQL procedures are best suited for:

* Batch operations involving multiple DML statements.
* Transactional logic that includes COMMIT and ROLLBACK.
* Use cases where returning a result set is not the primary requirement.

### Example Use Case

A procedure to update salaries for all employees in a department:

DELIMITER $$  
  
CREATE PROCEDURE UpdateSalaries(dept\_name VARCHAR(100), increment DECIMAL(10, 2))  
BEGIN  
 UPDATE employees  
 SET salary = salary + increment  
 WHERE department = dept\_name;  
END$$  
  
DELIMITER ;

## 5. When to Use PostgreSQL Functions

PostgreSQL functions are best suited for:

* Returning query results, such as rows or tables.
* Embedding logic in SQL queries, such as SELECT or WHERE.
* Use cases requiring deterministic logic (e.g., calculations, lookups).

### Example Use Case

A function to get employees’ names and salaries based on department:

CREATE OR REPLACE FUNCTION get\_department\_employees(dept\_name TEXT)  
RETURNS TABLE (name TEXT, salary NUMERIC) AS $$  
BEGIN  
 RETURN QUERY SELECT name, salary FROM employees WHERE department = dept\_name;  
END;  
$$ LANGUAGE plpgsql STABLE;

### Usage

SELECT \* FROM get\_department\_employees('Sales');

## 6. Why PostgreSQL Functions Are Better for SELECT ALL

* **Direct Result Return**:
  + PostgreSQL functions are specifically designed to return result sets, making them ideal for SELECT \* operations.
* **Integration**:
  + Functions can be embedded in SQL queries, making them more versatile.
* **Performance**:
  + Functions with STABLE or IMMUTABLE modifiers benefit from optimized execution plans, improving repeated query performance.

## Conclusion

* **MySQL Procedures**:
  + Best for procedural logic, transactional operations, or tasks involving multiple steps.
  + Less suitable for returning query results directly.
* **PostgreSQL Functions**:
  + Best for data retrieval and SELECT ALL operations.
  + More versatile, performant, and easier to integrate with SQL queries.

For a SELECT ALL operation, PostgreSQL functions are the better choice due to their ability to return result sets directly, integrate into queries, and benefit from execution plan caching.