PostgreSQL Stored Procedures

Stored procedures were introduced in PostgreSQL 11 to complement existing functions. A **stored procedure** is a database routine that encapsulates one or more SQL statements, typically for performing an action (e.g. batch update, data processing). Unlike functions, procedures do **not return a value** directly and are invoked with `CALL`. The key distinction is that procedures can issue transaction-control commands (`COMMIT`, `ROLLBACK`) inside them, whereas functions always execute within the caller's transaction and cannot commit or roll back. In practice, before v11 one simulated procedures by writing a `FUNCTION ... RETURNS void` and calling it; with v11+, the syntax `CREATE PROCEDURE` exists explicitly.

Differences from Functions:

- **Return value:** A function must include a `RETURNS` clause and return a result (scalar, row, or set). A procedure has no `RETURNS`; it always returns void to the caller. (Procedures can still use `OUT` or `INOUT` parameters to pass back data.)
- **Invocation:** Functions are called in SQL expressions (e.g. `SELECT myfunc(...)` or as part of a query). Procedures are executed *in isolation* using the `CALL` statement. For example:

```
CREATE PROCEDURE deactivate_unpaid_accounts()
LANGUAGE SQL
AS $$
  UPDATE accounts SET active = false WHERE balance < 0;
$$;

-- Call the procedure:
CALL deactivate_unpaid_accounts();
```

- **Transaction control:** Functions run within the existing transaction and cannot issue `COMMIT` or `ROLLBACK`. Procedures (if called outside any explicit transaction) **can** commit or roll back their own transactions. In a procedure, `COMMIT` ends the current transaction and immediately starts a new one. (A procedure called inside an active transaction will error)
- **Use case focus:** Functions are typically used to compute and return values for queries, whereas procedures are designed to perform tasks or workflows (e.g. batch jobs, administrative tasks) and then return control to the caller.

Syntax and Basic Usage

A stored procedure is defined with `CREATE PROCEDURE`. The syntax is similar to `CREATE FUNCTION` but without `RETURNS`. For example:

```
CREATE [ OR REPLACE ] PROCEDURE procedure_name (
    [ [ argmode ] argname argtype [ DEFAULT expr ] [, ...] ]
)
LANGUAGE plpgsql  -- or SQL, or other supported language
[ SECURITY { INVOKER | DEFINER } ]
AS $$
BEGIN
  -- SQL or PL/pgSQL code here
END;
$$;
```

- **Argument modes:** Use `IN` (default), `OUT`, or `INOUT` to define parameters. (Procedures do *not* support `OUT` by itself without a mode; use `INOUT` if needed.)
- **Language:** Commonly `plpgsql` (PostgreSQL's procedural language), or SQL for simple operations.
- **Security:** By default, procedures run with the caller's privileges (`SECURITY INVOKER`). You can mark a procedure `SECURITY DEFINER` to run with the owner's rights; note that a definer procedure **cannot** execute transaction control statements.
- **Example:** A procedure with IN/OUT parameters:

```
CREATE OR REPLACE PROCEDURE add_one(INOUT x integer)
LANGUAGE plpgsql
AS $$
BEGIN
  x := x + 1;
END;
$$;

-- Usage in psql:
DO $$
DECLARE
  val integer := 5;
BEGIN
  CALL add_one(val);
  RAISE NOTICE 'Result: %', val;  -- prints 6
END;
$$;
```

Call a procedure with `CALL`:

```
CALL procedure_name(arg1, arg2, ...);
```

In `psql`, any result values from `OUT` or `INOUT` parameters will be displayed as a result row. You cannot call a procedure inside a larger SQL expression or query; it stands alone.

To **manage** procedures, use standard commands: `DROP PROCEDURE procname(argtypes)`, `ALTER PROCEDURE`, and `CREATE OR REPLACE PROCEDURE`.

Procedures show up in schema listings similar to functions (e.g. use `\df` or query `pg_proc`). Remember that the procedure name resolution depends on argument types, so overloads are allowed.

Use Cases for Stored Procedures

Stored procedures are useful whenever encapsulating multi-statement logic on the server side is beneficial. Common scenarios include:

- **Modular business logic:** Encapsulate complex operations (e.g. order fulfillment, inventory adjustments) in the database to ensure consistency and reuse across applications. Breaking business logic into procedures promotes modular design and can enforce business rules close to the data.
- **Batch operations and ETL:** Perform bulk data transformations, imports, or maintenance tasks in one procedure call. For example, a procedure might clean up data in multiple tables in a single transaction, or loop through a set of inputs to insert/update records.
- **Data migrations and upgrades:** Use procedures to apply multi-step migration logic safely. Because procedures can include transaction controls, a migration procedure can commit parts of a migration in stages. This is especially helpful for complex upgrades (e.g. splitting a table, migrating legacy data) that might require conditional commits.
- **Data validation and integrity:** Encapsulate validation checks inside procedures called by triggers or application code. A procedure can examine data, raise errors on bad input, or apply fixes. For example, a data-import procedure could verify and reject invalid rows in a batch.
- **Scheduling tasks:** Combine procedures with job schedulers (such as `cron`, `pgAgent`, or an external scheduler) to run routine tasks (backups, report generation, cleanup) at set intervals. Scheduling simply calls a procedure at the desired time, leveraging its transaction control.
- **Complex query orchestration:** Sequence multiple queries or operations that must run together. For instance, a procedure might open and fetch from cursors, loop over results, and conditionally apply updates.
- **Security and abstraction:** Expose procedures as a controlled API for certain operations. By granting `EXECUTE`on a procedure (and perhaps using `SECURITY DEFINER`), you can allow certain users to perform actions without giving them full table access.

Example use case – batch update of two tables atomically:

```
CREATE PROCEDURE transfer_stock(from_loc text, to_loc text, item_id int, qty int)
LANGUAGE plpgsql
AS $$
BEGIN
  UPDATE stock
    SET quantity = quantity - qty
    WHERE location = from_loc AND item = item_id;
  UPDATE stock
    SET quantity = quantity + qty
    WHERE location = to_loc AND item = item_id;
  IF NOT FOUND THEN
    RAISE EXCEPTION 'Invalid stock transfer';
  END IF;
  COMMIT;  -- finalize transaction mid-procedure
END;
$$;
```

Then an external job can call `CALL transfer_stock('A','B',42,10);` to move items. The procedure's own `COMMIT` ensures the transfer is saved even if not in a larger transaction.

Edge Cases and Tricky Behaviors

While powerful, procedures have important caveats:

- **Transaction boundaries:** A procedure can **only** execute `COMMIT` or `ROLLBACK` if it was called outside any active transaction. If you try `BEGIN; CALL proc(); COMMIT;`, PostgreSQL will error (because the `CALL` starts a new transaction internally). In practice, clients should ensure they are in auto-commit mode (or explicitly `SET AUTOCOMMIT=on`) when calling a transaction-controlling procedure. The official JDBC example shows turning `autoCommit=true` before calling such a procedure.
- **SECURITY DEFINER limitations:** A procedure marked `SECURITY DEFINER` (running with owner's rights) **cannot** execute transaction control statements. If your definer procedure contains `COMMIT`/`ROLLBACK`, it will fail.
- **Error handling:** By default, an unhandled error in a procedure aborts the current transaction (as in any PL/pgSQL function). You can catch exceptions using `BEGIN ... EXCEPTION ... END;` blocks. However, note that each procedure call runs in a single transaction scope (unless you COMMIT mid-way). So if an error is caught and handled, the procedure can continue, but if uncaught it bubbles to the caller.
- **No return value:** A procedure never returns a query result. If you need to return rows, you must use `OUT` parameters or open cursors. (For example, a procedure could declare a `refcursor` OUT parameter, open it to a query, and the caller can fetch from it.) Unlike some other DBMS, you cannot `SELECT * FROM proc(...)` in PostgreSQL.
- **Cannot be used in queries or triggers:** You cannot call a procedure inside a `SELECT` or as a function in a trigger (triggers must call functions). Procedures stand alone.

- **Driver quirks:** Some client libraries or JDBC drivers may default to treating calls as functions. For PostgreSQL's JDBC driver, you may need to enable escape syntax for calls. For example, setting `escapeSyntaxCallMode=callIfNoReturn` in the JDBC URL allows using `{call proc_name(?,?)}`. The pgJDBC documentation example explicitly sets this property and uses `prepareCall("{call ...}")`.
- **Portability and versioning:** Procedures are PostgreSQL-specific. Logic in procedures must be maintained separately (e.g. in migration scripts or source control) since the database itself doesn't version them. Changing procedure code requires `CREATE OR REPLACE`, and previous definitions are overwritten. This can be an "edge case" when rolling back changes if no scripts are kept.
- **Permissions:** Grant `EXECUTE` on a procedure to allow other roles to call it. The invoker needs privileges on any objects used inside the procedure unless it's defined as SECURITY DEFINER.

By being aware of these details, you can avoid subtle bugs (e.g. failed transactions or permission errors). In summary, ensure that any `CALL` of a transaction-managing procedure is done outside an open `BEGIN...END` block, and watch out for security and return-value implications.

Performance and Optimization

Stored procedures mainly move work into the database server. Performance considerations include:

- **Network round-trips:** Procedures can improve overall efficiency by reducing back-and-forth between client and server. A batch of operations in one `CALL` is faster than multiple separate queries from the client.
- **Plan caching (PL/pgSQL):** In PL/pgSQL procedures, SQL statements are parsed and planned on first execution in a session and cached for reuse. This can modestly speed up repeated calls. (By contrast, a standalone SQL sent repeatedly from the client is re-planned each time.) However, if each call uses different schemas or query text, the cached plan is invalidated and recompilation is needed.
- **Resource usage:** Because stored procedures execute on the server, heavy computations increase database CPU/memory load. Ensure the DB server is provisioned accordingly. Use set-based SQL inside procedures where possible (avoid row-by-row loops if a single UPDATE could do the job).
- **Prepared Statements / Queries:** Inside plpgsql, you can still use `EXECUTE` for dynamic SQL. Be mindful that generic plans (with unspecified parameters) may be used if plpgsql cannot tailor a plan, which can impact performance for skewed data. In critical cases, consider writing performance-sensitive logic in a native language (e.g. C) or using more specific queries.
- **Work_mem and resources:** For operations like sorting or hashing inside procedures, PostgreSQL memory settings (work_mem, maintenance_work_mem) still apply. For large data processing, you may need to tune these or break work into smaller chunks.
- **Indexing and query tuning:** Procedures are not magic – normal performance best practices (proper indexes, avoiding full table scans when possible) still apply to any SQL they run. Use `EXPLAIN` on the queries inside the procedure to ensure they use indexes and efficient plans.

- **Transactions:** Long-running transactions (procedures that do a lot before COMMIT) can hold locks and bloat WAL. If possible, break very large operations into smaller chunks or commits to avoid bloating transaction logs or blocking other queries.

In general, stored procedures trade developer convenience and reduced latency for increased work on the server. PostgreSQL's JIT compilation (in 11+) may also apply to complex queries inside plpgsql, but profiling individual workloads is recommended. As one analysis notes, performance gains from plan-caching in functions/procedures are usually small; focus on simplicity and correctness first.

Calling Procedures from Java

PostgreSQL procedures can be invoked from Java just like functions, with a few specifics. Below are examples using plain JDBC, Spring JDBC, and Spring Data JPA.

Plain JDBC (Driver)

Use `CallableStatement` or standard `Statement` with the `CALL` syntax. For example, the PostgreSQL JDBC documentation shows this pattern for a procedure with an INOUT parameter and a `COMMIT` inside:

```
// Establish connection with escape syntax for calls
String url = "jdbc:postgresql://localhost/test";
Properties props = new Properties();
props.setProperty("escapeSyntaxCallMode", "callIfNoReturn");  // enable
CALL
Connection con = DriverManager.getConnection(url, props);

// Ensure auto-commit is ON, since procedure will COMMIT internally
con.setAutoCommit(true);

// Create and call the procedure
CallableStatement proc = con.prepareCall("{call commitproc(?)}");
proc.setInt(1, 100);
proc.execute();
proc.close();
```

This corresponds to a PostgreSQL setup like:

```
CREATE OR REPLACE PROCEDURE commitproc(INOUT a bigint)
LANGUAGE plpgsql AS $$
BEGIN
  INSERT INTO temp_val VALUES (a);
  COMMIT;
END;
$$;
```

Key points:

- Use `{call proc_name(?, ?)}` with `prepareCall`.
- Register out parameters if needed (e.g. `proc.registerOutParameter(1, Types.BIGINT)`).

- Set `autoCommit=true` on the JDBC `Connection` so that the `CALL` is not inside another transaction.
- Starting with postgresql JDBC driver 42.x, enabling the escape syntax call mode (e.g. `callIfNoReturn`) allows calling procedures; otherwise the driver may think it's a function and give an error.