Database Design and Programming

Tutorial 6: Stored Procedures and Triggers

Biswadeep Sen
School of Computing
National University of Singapore
biswadeep@u.nus.edu

Stored Functions vs Stored Procedures

Function (`CREATE FUNCTION`)

Returns a value ✅.

Callable in SQL expressions — e.g., in `SELECT`, `WHERE`, `ORDER BY`, joins, views.

Example: `SELECT my_fn(col) AS x FROM t;`

Procedure (`CREATE PROCEDURE`)

Returns no value ⛔.

Run as a standalone command: `CALL my_proc(...);`

Use `RAISE NOTICE/EXCEPTION` for messages/errors.

👍**RULE OF THUMB:**

Do I need the output *inside a query* (SELECT/WHERE/ORDER BY/join/view/index/constraint)? → Yes ⇒ FUNCTION.

Am I primarily *changing data* (INSERT/UPDATE/DELETE) as a task the app/user runs without returning a value? → Yes ⇒ PROCEDURE.

1.(a) Write a **function/procedure** named **borrow_book** with parameters
email VARCHAR(256), **isbn13 CHAR(14)**, and **borrow_date DATE**.
It should **check** whether there is an **available copy** of the book; if **available**,
insert a new loan record for the borrower. Return a message indicating **success**
or **failure**.

Then **execute** this scenario using your **borrow_book** function/procedure:
Adeline Wong (**awong007@msn.com**) tries to **borrow 3 copies** of **"Applied Calculus"** by
Deborah Hughes-Hallett et al., **ISBN13 978-0470170526**.

Let's do this using a stored function first!

```
CREATE OR REPLACE FUNCTION borrow_book (
   borrower_email VARCHAR(256), isbn13 CHAR(14), borrow_date DATE
) RETURNS TEXT AS $$

DECLARE
   available_copy RECORD;
BEGIN
   SELECT * INTO available_copy
   FROM copy c
   WHERE c.book = isbn13
      AND NOT EXISTS (
         SELECT 1 FROM loan l
         WHERE l.book = c.book
            AND l.copy = c.copy
            AND l.owner = c.owner
            AND l.returned IS NULL
      )
   LIMIT 1;
   IF NOT FOUND THEN
      RETURN 'No available copies of the book with ISBN13 : '  ||
isbn13;
   ELSE
      INSERT INTO loan (borrower, owner, book, copy, borrowed)
      VALUES (borrower_email, available_copy.owner,
              available_copy.book, available_copy.copy, borrow_date);
      RETURN 'Book with ISBN13 : ' || isbn13 ||
             ' has been successfully borrowed by '  || borrower_email;
   END IF;
END;
$$ LANGUAGE plpgsql;
```

Header

Find

Decide

Insert

Return

```
CREATE OR REPLACE FUNCTION borrow_book (
  borrower_email VARCHAR(256), isbn13 CHAR(14), borrow_date DATE
) RETURNS TEXT AS $$

DECLARE
  available_copy  RECORD;
BEGIN
  SELECT * INTO available_copy
  FROM copy c
  WHERE c.book = isbn13
    AND NOT EXISTS (
      SELECT 1 FROM loan l
      WHERE l.book = c.book
        AND l.copy = c.copy
        AND l.owner = c.owner
        AND l.returned IS NULL
    )
  LIMIT 1;
  IF NOT FOUND THEN
    RETURN 'No available copies of the book with ISBN13 : '  ||
isbn13;
  ELSE
    INSERT INTO loan (borrower, owner, book, copy, borrowed)
    VALUES (borrower_email, available_copy.owner,
            available_copy.book, available_copy.copy, borrow_date);
    RETURN 'Book with ISBN13 : ' || isbn13 ||
           ' has been successfully borrowed by '  || borrower_email;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

📌 *NOTE:*

Header: inputs → TEXT

Find: pick one free copy (fills `available_copy`)

Decide: `IF NOT FOUND` → return "no copy"; otherwise `INSERT` the loan.

Insert: add row to `loan`

Return: success message

Header

Find

Decide

Insert

Return

```sql
CREATE OR REPLACE FUNCTION borrow_book (
  borrower_email VARCHAR(256), isbn13 CHAR(14), borrow_date DATE
) RETURNS TEXT AS $$

DECLARE
  available_copy RECORD;
BEGIN
  SELECT * INTO available_copy
  FROM copy c
  WHERE c.book = isbn13
    AND NOT EXISTS (
      SELECT 1 FROM loan l
      WHERE l.book = c.book
        AND l.copy = c.copy
        AND l.owner = c.owner
        AND l.returned IS NULL
    )
  LIMIT 1;
  IF NOT FOUND THEN
    RETURN 'No available copies of the book with ISBN13 : '  ||
isbn13;
  ELSE
    INSERT INTO loan (borrower, owner, book, copy, borrowed)
    VALUES (borrower_email, available_copy.owner,
            available_copy.book, available_copy.copy, borrow_date);
    RETURN 'Book with ISBN13 : ' || isbn13 ||
           ' has been successfully borrowed by '  || borrower_email;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

📌 *NOTE:*

RECORD: one-row container; fields come from the SELECT.

SELECT … INTO: Assign the first matching row to `available_copy`; sets FOUND.

FOUND: Built-in PL/pgSQL flag—set by the last SQL; for SELECT INTO: **TRUE** if a row fetched, else **FALSE**.

LIMIT 1: stop at first match (faster, clearer).

NOT EXISTS (… returned IS NULL): copy is **not** currently on loan.

```
CREATE OR REPLACE FUNCTION borrow_book (
  borrower_email VARCHAR(256), isbn13 CHAR(14), borrow_date DATE
) RETURNS TEXT AS $$ -- function header: takes (email, ISBN, date) and
                        returns a TEXT message
DECLARE
  available_copy RECORD;          -- generic one-row container
BEGIN
  SELECT * INTO available_copy    -- Find ONE free copy of this ISBN
  FROM copy c
  WHERE c.book = isbn13
    AND NOT EXISTS (              -- no active loan for this copy
      SELECT 1 FROM loan l
      WHERE l.book = c.book
        AND l.copy = c.copy
        AND l.owner = c.owner
        AND l.returned IS NULL
    )
  LIMIT 1;                        -- only need one for FOUND to be TRUE
  IF NOT FOUND THEN               -- SELECT INTO got no row
    RETURN 'No available copies of the book with ISBN13 : '  || isbn13;
  ELSE                            -- insert borrow into loan
    INSERT INTO loan (borrower, owner, book, copy, borrowed)
    VALUES (borrower_email, available_copy.owner,
            available_copy.book, available_copy.copy, borrow_date);
    RETURN 'Book with ISBN13 : ' || isbn13 ||
           ' has been successfully borrowed by '  || borrower_email;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION borrow_book (
  borrower_email VARCHAR(256), isbn13 CHAR(14), borrow_date DATE
) RETURNS TEXT AS $$

DECLARE
  available_copy RECORD;
BEGIN
  SELECT * INTO available_copy
  FROM copy c
  WHERE c.book = isbn13
    AND NOT EXISTS (
      SELECT 1 FROM loan l
      WHERE l.book = c.book
        AND l.copy = c.copy
        AND l.owner = c.owner
        AND l.returned IS NULL
    )
  LIMIT 1;
  IF NOT FOUND THEN
    RETURN 'No available copies of the book with ISBN13 : '  || isbn13;
  ELSE
    INSERT INTO loan (borrower, owner, book, copy, borrowed)
    VALUES (borrower_email, available_copy.owner,
            available_copy.book, available_copy.copy, borrow_date);
    RETURN 'Book with ISBN13 : ' || isbn13 ||
           ' has been successfully borrowed by '  || borrower_email;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

1. No loan rows at all → passes (copy is free) ✅
2. Only past loans where `returned` is NOT NULL → passes (was returned) ✅
3. At least one loan row with `returned IS NULL` → fails (still out) ❌

(1.a) Continued ….

```
--Invocation
SELECT borrow_book ('awong007@msn.com', '978-0470170526' ,CURRENT_DATE);
SELECT borrow_book ('awong007@msn.com', '978-0470170526' ,CURRENT_DATE);
SELECT borrow_book ('awong007@msn.com', '978-0470170526' ,CURRENT_DATE);
```

1.(a) Write a **function/procedure** named **borrow_book** with parameters **email VARCHAR(256)**, **isbn13 CHAR(14)**, and **borrow_date DATE**.
It should **check** whether there is an **available copy** of the book; if **available**, insert a new loan record for the borrower. Return a message indicating **success** or **failure**.

Then **execute** this scenario using your **borrow_book** function/procedure:
Adeline Wong (**awong007@msn.com**) tries to **borrow 3 copies** of **"Applied Calculus"** by **Deborah Hughes-Hallett et al.**, **ISBN13 978-0470170526**.

Now let's do this using a stored procedure!

```
CREATE OR REPLACE PROCEDURE borrow_book_proc(
   borrower_email VARCHAR(256), isbn13 CHAR(14), borrow_date DATE
   ) AS $$
DECLARE available_copy RECORD;
BEGIN
  SELECT * INTO available_copy FROM copy c
    WHERE c.book = isbn13
        AND NOT EXISTS(
           SELECT 1 FROM loan l
           WHERE l.book=c.book
              AND l.copy=c.copy AND l.owner=c.owner
              AND l.returned IS NULL
  )
  LIMIT 1;
  IF NOT FOUND
  THEN
    RAISE NOTICE 'No available copies of the book with ISBN13:%’ , isbn13;
    RETURN;
  ELSE
    INSERT INTO loan(borrower, owner, book, copy, borrowed)
    VALUES(borrower_email, available_copy.owner, available_copy.book,
           available_copy.copy, borrow_date);
    RAISE NOTICE 'Book with ISBN13: % has been successfully borrowed by
                                 %' , isbn13, borrower_email;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE PROCEDURE borrow_book_proc(
   borrower_email VARCHAR(256), isbn13 CHAR(14), borrow_date DATE
   ) AS $$
DECLARE available_copy RECORD;
BEGIN          -- Check for a copy of the book that is not currently borrowed
  SELECT * INTO available_copy FROM copy c
    WHERE c.book = isbn13
        AND NOT EXISTS(
           SELECT 1 FROM loan l
           WHERE l.book=c.book
              AND l.copy=c.copy AND l.owner=c.owner
              AND l.returned IS NULL
  )
  LIMIT 1;
  IF NOT FOUND -- Raise notice if no available copy found
  THEN
    RAISE NOTICE 'No available copies of the book with ISBN13:%' , isbn13;
    RETURN;     -- Return just exits here
  ELSE          -- If copy available insert to loan
    INSERT INTO loan(borrower, owner, book, copy, borrowed)
    VALUES(borrower_email, available_copy.owner, available_copy.book,
           available_copy.copy, borrow_date);
    RAISE NOTICE 'Book with ISBN13: % has been successfully borrowed by
                                    %' , isbn13, borrower_email;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

1(a) continued ……

```
--Invocation
CALL borrow_book_proc ('awong007@msn.com', '978-0470170526' ,CURRENT_DATE);
CALL borrow_book_proc ('awong007@msn.com', '978-0470170526' ,CURRENT_DATE);
CALL borrow_book_proc ('awong007@msn.com', '978-0470170526' ,CURRENT_DATE);
```

2. In our current database, **Adeline Wong (awong007@msn.com)** has borrowed 6 books and has not returned any.

We want an additional constraint: a student may **borrow up to 6 books** at a time. In other words, if a student already has 6 unreturned books, **they cannot borrow another.**

We will explore two strategies to enforce this constraint.

Triggers

Definition: A **trigger** is a procedure or function executed when a database event (**`INSERT/ UPDATE/ DELETE etc.`**) occurs on a table.

Why they are used:

- Enforce **data integrity** and business rules.
- **Propagate/repair** changes (e.g., audit rows).

2. (a) **Create a trigger** that **checks** when a student tries to borrow a copy of a book; the loan succeeds only if that student **does not already have 6 active loans**.

(2.a) Continued…

```
CREATE OR REPLACE FUNCTION check_local_loan_limit()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    active_loan_count  INT;
BEGIN

    SELECT COUNT(*)INTO active_loan_count
    FROM loan l
    WHERE l.borrower = NEW.borrower
      AND l.returned IS NULL;

    IF active_loan_count >= 6
    THEN
      RETURN NULL;
    ELSE
      RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

(2.a) Continued…

```
CREATE OR REPLACE FUNCTION check_local_loan_limit()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    active_loan_count  INT;
BEGIN
    -- Count active (unreturned) loans for this borrower
    SELECT COUNT(*) INTO active_loan_count
    FROM loan l
    WHERE l.borrower = NEW.borrower
      AND l.returned IS NULL;

    IF active_loan_count >= 6
    THEN
      RETURN NULL;    -- prevent borrowing
    ELSE
      RETURN NEW;     -- allow borrowing
    END IF;
END;
$$ LANGUAGE plpgsql;
```

📌 *NOTE:*
Why INT (not RECORD): `COUNT(*)` gives **one number**. An INT stores that number directly.

NEW → the row being inserted.

(2.a) Continued…

```
-- Create trigger: enforce local loan limit on inserts/updates to loan
CREATE TRIGGER enforce_local_loan_limit_insert
BEFORE INSERT ON loan
FOR EACH ROW EXECUTE FUNCTION check_local_loan_limit();

-- Test the trigger (assuming a PROCEDURE exists)
CALL borrow_book_proc('awong007@msn.com', '978-1449389673',CURRENT_DATE);

-- Drop the trigger and its function (for cleanup / re-tests)
DROP TRIGGER enforce_local_loan_limit_insert ON loan;
DROP FUNCTION check_local_loan_limit();
```

📌 *NOTE:*
This trigger is **designed for BEFORE INSERT only**. It counts active loans for `NEW.borrower`; if the count ≥ 6 → RETURN NULL (block the insert), else → RETURN NEW.

Updates not covered
Using this as-is on UPDATE is wrong—you must add update logic that handles (1) **reactivation** (`returned:` NOT NULL → NULL) and (2) **borrower changes**. Treat that as a small homework!

Triggers: Components

It has two components:

- **Trigger**: the binding to a table/view and event (and timing: **BEFORE/AFTER**).
- **Trigger function**: the **action** to run when the event occurs.

```
-- Create trigger: enforce global loan limit on inserts/updates to loan
CREATE TRIGGER enforce_local_loan_limit_insert  -- Trigger
BEFORE INSERT ON loan
FOR EACH ROW EXECUTE FUNCTION check_local_loan_limit(); --Trigger function
```

Where the BEFORE INSERT (local) trigger falls short

⚠️ **What it can *miss* (*non-exhaustive*):**

- Updates that **“unreturn”** a book (`returned: non-NULL → NULL`).
  If a borrower already has 6 active loans, this **can push them to 7** and slip past if you only trigger on INSERT.

- Updates that **change the borrower** on an existing row (reassigning a loan) can also bump someone over 6.

🛠️ **Fix:**

- **BEFORE UPDATE (local):** Write a dedicated `BEFORE UPDATE` trigger (or extend the function) that handles *reactivations* (`returned: NOT NULL → NULL`) and *borrower changes*. Don’t reuse the INSERT-only trigger—its logic will block harmless edits and miss real cases.

2. (b) Create a trigger to check that **no** student has **more than 6 active loans.**

2 (b). Continued…..

```
CREATE OR REPLACE FUNCTION check_global_loan_limit()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    violating_student  RECORD;
BEGIN
    SELECT l.borrower INTO violating_student
    FROM loan l
    WHERE l.returned IS NULL
    GROUP BY l.borrower
    HAVING COUNT(*) > 6
     LIMIT 1;

    IF violating_student IS NOT NULL THEN
        RAISE EXCEPTION '% has borrowed more than 6 books' ,
            violating_student.borrower;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

📌 *NOTE:*

`WHERE l.returned IS NULL` → keep only **active (unreturned)** loans.

`GROUP BY l.borrower` → collapse those rows into one group **per borrower email**.

`HAVING COUNT(*) > 6` → keep only groups whose **active-loan count > 6** (violators).

2 (b). Continued…..

```
CREATE OR REPLACE FUNCTION check_global_loan_limit()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    violating_student RECORD;    -- will hold one borrower if any violates
BEGIN
    SELECT l.borrower INTO violating_student
    FROM loan l
    WHERE l.returned IS NULL    -- active loans only
    GROUP BY l.borrower
    HAVING COUNT(*) > 6
     LIMIT 1;    -- pick any one violator if many

    IF violating_student IS NOT NULL THEN
        RAISE EXCEPTION '% has borrowed more than 6 books' ,
            violating_student.borrower;
    END IF;

    RETURN NEW;   -- allow the row if no violation
END;
$$ LANGUAGE plpgsql;
```

2 (b). Continued…..

```
-- Create trigger: enforce global loan limit on inserts/updates to loan
CREATE TRIGGER enforce_global_loan_limit
AFTER INSERT OR UPDATE ON loan -- Same trigger handles both
FOR EACH ROW
EXECUTE FUNCTION check_global_loan_limit();

-- Test the trigger (assuming a PROCEDURE exists)
CALL borrow_book_proc('awong007@msn.com','978-1449389673',CURRENT_DATE);

-- Drop the trigger and its function (for cleanup / re-tests)
DROP TRIGGER enforce_global_loan_limit ON loan;
DROP FUNCTION check_global_loan_limit();
```

📌 *NOTE:*

How it works: After each INSERT/UPDATE, scan `loan`. If anyone has **> 6** active loans → **RAISE EXCEPTION** (undo / rollback). Else → **RETURN NEW**.

The **same code (trigger)** can be used for both insertion and update in this case!

Thank you for joining!

Got questions? Post them on the forum or email me:
biswadeep@u.nus.edu
(I reply **within 2 working days** — *faster if coffee is strong* ☕)

Because your learning matters to me! 😊

NUS
National University of Singapore