Programming: Stored Procedures and Triggers

Students at the National University of Ngendipura (NUN) buy books for their studies. They also lend and borrow books to and from other students. Your company, Apasaja Private Limited, is commissioned by NUN Students Association (NUNStA) to implement an online book exchange system that records information about students, books that they own and books that they lend and borrow.

The database records the name, faculty, and department of each student. Each student is identified in the system by her email. The database also records the date at which the student joined the university (year attribute).

The database records the title, authors, publisher, year and edition and the ISBN-10 and ISBN-13 for each book. The International Standard Book Number, ISBN-10 or -13, is an industry standard for the unique identification of books. It is possible that the database records books that are not owned by any students (because the owners of a copy graduated or because the book was advised by a lecturer for a course but not yet purchased by any student.)

The database records the date at which a book copy is borrowed and the date at which it is returned. We refer to this information as a loan record.

For auditing purposes the database records information about the books, the copies and the owners of the copies as long as the owners are students or as there are loan records concerning the copies. For auditing purposes the database records information about graduated students as long as there are loan records concerning books that they owned.

This tutorial uses the schema and data for the database created in "SQL: Creating and Populating Tables" including all the updates done during the tutorial.

Questions

Not all questions will be discussed during tutorial. You are expected to attempt them before coming to the tutorial. You may be randomly called to present your answer during tutorial. You are encouraged to discuss them on Canvas Discussion.

1. **Stored Functions and Procedures**.
   (a) Write a function/procedure `borrow_book_func` that, given the the email of a borrower (`VARCHAR(256)`), the ISBN13 of a book (`CHAR(14)`), and the borrow date (`DATE`), checks whether there is an available copy of the book, and, if that is the case, inserts

a new loan record of the copy by the borrower. Return a message indicating success or failure of insertion.

Additionally, execute the following scenario using your function.

> Adeline Wong, with email awong007@msn.com, tries to borrow 3 copies of "Applied Calculus" by Deborah Hughes-Hallett, et al. with ISBN13 value of 978-0470170526.

Comments:

One possible function is the following.

Code: Function

```
CREATE OR REPLACE FUNCTION borrow_book_func (
  borrower_email VARCHAR (256), isbn13 CHAR (14), borrow_date DATE
) RETURNS TEXT AS $$
DECLARE
  available_copy RECORD ;
BEGIN
  -- Check for a copy of the book that is not currently borrowed
  --   (i.e., no active loan)
  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 -- No available copy found, return a message
  THEN
    RETURN 'No available copies of the book with ISBN13 : ' || isbn13;
  ELSE -- An available copy found
    -- Insert a new record into the loan table to record the borrowing
    INSERT INTO loan (borrower, owner, book, copy, borrowed)
    VALUES (borrower_email, available_copy.owner,
      available_copy.book, available_copy.copy, borrow_date);
    -- Return a success message
    RETURN 'Book with ISBN13 : ' || isbn13 ||
      ' has been successfully borrowed by ' || borrower_email;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

Code: Invocation

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

Comments:

One possible procedure is the following.

Code: Procedure

```
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
  --   (i.e., no active loan)
  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 ISNULL
    )
  LIMIT 1;

  IF NOT FOUND -- No available copy found, raise notice
  THEN
    RAISE NOTICE 'No available copies of the book with ISBN13 : %',
      isbn13;
    RETURN;
  ELSE -- An available copy found
    -- Insert a new record into the loan table to record the borrowing
    INSERT INTO loan (borrower, owner, book, copy, borrowed)
    VALUES (borrower_email, available_copy.owner,
      available_copy.book, available_copy.copy, borrow_date);
    -- Raise a success message
    RAISE NOTICE 'Book with ISBN13 : % has been successfully
      borrowed by %', isbn13, borrower_email;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

Note: In cases of failure, you can also `RAISE EXCEPTION` to stop any further execution. You may then choose to trap [3] the exception (equivalent to a `catch` clause) and recover from it; otherwise, the exception will end the current transaction, and the database is rolled back to the state before the transaction begins.

Code: Invocation

```
CALL borrow_book_proc ('awong007@msn.com', '978-0470089156',
  CURRENT_DATE);
CALL borrow_book_proc ('awong007@msn.com', '978-0470089156',
  CURRENT_DATE);
CALL borrow_book_proc ('awong007@msn.com', '978-0470089156',
  CURRENT_DATE);
CALL borrow_book_proc ('awong007@msn.com', '978-0470089156',
  CURRENT_DATE);
```

Comments:

Stored functions/procedures follow a different paradigm from SQL queries. In SQL queries, we describe **what** is the data we want to have without specifying how to compute it. On the other hand, in stored functions, we describe **how** to compute the data.

The language `plpgsql` has all the necessary constructs to compute any possible computation that can be done with other languages. In particular, we can do selection (e.g., `IF` - `THEN` - `ELSE` - `END IF`) and we can do repetition (e.g., `LOOP` - `END LOOP`).

However, a typical repetition has the structure of `LOOP` - `END LOOP` with the use of explicit `EXIT WHEN NOT FOUND` in the middle. This is closer to the following in a C-like language.

```
while(true) { if (not found) { break; }}
```

`plpgsql` is close to the BASIC language family. A common feature is the use of `END` to close a block as opposed to the use of braces (i.e., `{ ... }`) in C-like languages or indentation in Python.

2. **Triggers**.

   In our current database, Adeline Wong, with email `awong007@msn.com`, already borrowed 6 books and has not returned any of the books.

   We would like to introduce an additional constraint: A student may only borrow up to 6 books at a time. In other words, if a student has 6 books that have not been returned yet, the student cannot borrow another book.

   Let us explore two different strategies to enforce this constraint.

   (a) Create a trigger that checks if a student is trying to borrow copy of a book, the loan is only successful if that student does not already have 6 active loans.

Comments:

This strategy can be classified as a "local" strategy as it checks for local consistency (i.e., the current student being inserted/updated is not violating the condition).

Below is the trigger for `INSERT`. Since we use `RETURN NULL` in the trigger function to prevent insertion, we have to provide a `BEFORE` trigger. We can also use `RAISE EXCEPTION`. In that case, we can use either a `BEFORE` or an `AFTER` trigger.

Code: Trigger Function

```
CREATE OR REPLACE FUNCTION check_local_loan_limit()
RETURNS TRIGGER AS $$
DECLARE
  active_loan_count INT;
BEGIN
  -- Count the number of active loans (not yet returned)
  SELECT COUNT(*) INTO active_loan_count
  FROM loan l
  WHERE l.borrower = NEW.borrower
    AND l.returned ISNULL;

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

Code: INSERT Trigger

```
CREATE TRIGGER enforce_local_loan_limit_insert
BEFORE INSERT ON loan
FOR EACH ROW EXECUTE FUNCTION check_local_loan_limit();
```

The trigger for `UPDATE` and potentially a new trigger function is left as an exercise. Think about all `UPDATE` queries that may potentially break the constraint. Some ideas are shown at the end of this question, but the list is non exhaustive.

Code: Test the Trigger

```
CALL borrow_book_proc('awong007@msn.com', '978-1449389673',
  CURRENT_DATE);
```

To drop the trigger in order to test other triggers:

Code: Drop Trigger

```
DROP TRIGGER enforce_local_loan_limit_insert ON loan;
DROP FUNCTION check_local_loan_limit();
```

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

Comments:

This strategy can be classified as a "global" strategy as it checks for global consistency (i.e., no student is violating this).

Code: Trigger Function

```
CREATE OR REPLACE FUNCTION check_global_loan_limit()
RETURNS TRIGGER AS $$
DECLARE
  violating_student RECORD;
BEGIN
  -- Check if there is any student with more than 6 active loans
  SELECT l.borrower INTO violating_student
  FROM loan l
  WHERE l.returned ISNULL
  GROUP BY l.borrower
  HAVING COUNT(*) > 6;

  IF violating_student IS NOT NULL
  THEN -- There is a violation, raise exception
    RAISE EXCEPTION '% has borrowed more than 6 books',
      violating_student;
  ELSE
    RETURN NEW;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

In this case, we can use the same trigger function for both `INSERT` and `UPDATE` triggers. Since the check has to be done after all modifications are made, we can only use an `AFTER` trigger. This also means that we can only use `RAISE EXCEPTION` to prevent any modification to the database.

Code: Trigger

```
CREATE TRIGGER enforce_global_loan_limit
AFTER INSERT OR UPDATE ON loan
FOR EACH ROW EXECUTE FUNCTION check_global_loan_limit();
```

Is there any other table we need to attach a trigger to? We can easily check by looking at the query that would be used to test the trigger. Here, only the table `loan` is used in our checking query. So, we only need a trigger for the table `loan`.

Code: Test the Trigger

```
CALL borrow_book_proc('awong007@msn.com', '978-1449389673',
  CURRENT_DATE);
```

To drop the trigger in order to test other triggers:

Code: Drop Trigger

```
DROP TRIGGER enforce_global_loan_limit ON loan;
DROP FUNCTION check_global_loan_limit();
```

Comments:

We note in conclusion that there are many different ways to enforce the same constraints. Some constraints may not even require a trigger. This is mainly due to the diverse and sometimes redundant constructs.

As mentioned in the comments for Question 2a, the provided `INSERT` trigger is not a complete solution yet; i.e., it does not fully enforce the constraint. Some of the other scenarios that should also be handled by Question 2a are listed below.

- Insert by means of an `INSERT INTO` statement or by a stored function for Adeline Wong.

  ⇒ Not inserted.
- Insert by means of an `INSERT INTO` statement or by a stored function for a student that only has 6 active loans.

  ⇒ Inserted.
- Update books that are already returned to "unreturn" them (i.e., modify attribute `returned` from a non-`NULL` value to `NULL` value) for Adeline Wong.

  ⇒ Not updated.
- Update books that are not yet returned to be returned (i.e., modify attribute `returned` from a `NULL` value to non-`NULL` value) for Adeline Wong.

  ⇒ Updated.
- Update books that are already returned to "unreturn" them (i.e., modify attribute `returned` from a non-`NULL` value to `NULL` value) for a student that only has 5 active loans.

  ⇒ Updated.
- Update books that are not yet returned to be returned (i.e., modify attribute `returned` from a `NULL` value to non-`NULL` value) for a student that only has 5 active loans.

  ⇒ Updated.

There are advantages and disadvantages to both approaches. We encourage students to experiment and write alternative code.

Note that triggers and trigger functions are not deleted when the underlying tables are deleted. The triggers may reattach to the new table with the same name. To ensure that you have deleted all stored functions as well as triggers, we recommend dropping the `SCHEMA` and recreating it.

Code: DROP SCHEMA

```
-- change `public` to the name of your schema
DROP SCHEMA public CASCADE;
CREATE SCHEMA public;
```

References

[1] S. Bressan and B. Catania. *Introduction to Database Systems*. McGraw-Hill Education, 2006. ISBN: 9780071246507.

[2] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. 2nd ed. Prentice Hall Press, 2008. ISBN: 9780131873254.

[3] *PostgreSQL Docs: Trapping Errors*. `https://www.postgresql.org/docs/current/plpgsql-control-structures.html#PLPGSQL-ERROR-TRAPPING`. [Online; last accessed 2025].

[4] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. 2nd. USA: McGraw-Hill, Inc., 2000. ISBN: 0072440422.

[5] *W3schools Online Web Tutorials*. `https://www.w3schools.com/`. [Online; last accessed 2025].