



Tutorial: 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 “Tutorial: Creating and Populating Tables”, including all the updates done during that tutorial and the previous tutorial “Tutorial: Stored Procedures”. You may also use the stored functions and procedures defined in “Tutorial: Stored Procedures”.

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. 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 1a, 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 1a 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;
```

2. Update.

When a student graduates, we update the `graduate` attribute in the table `student`. The only check currently is `CHECK(graduate >= year)`. We would also like to enforce that the graduation date is not in the future. This can be enforced with another `CHECK` constraint (i.e., `CHECK(graduate <= CURRENT_DATE)`).

However, the drawback of such a `CHECK` constraint is that an incorrect update will be rejected with an error. We want to avoid errors as much as possible. To that end, let us create a trigger that can handle violating updates.

- (a) Create a trigger that, if the `graduate` date is in the future, modifies the `graduate` date to the current date, therefore making the operation valid. The trigger should be invoked for insertion or update.

Comments:

Note that using the “global” strategy, we can only prevent insertion or update. Hence, we have to use the “local” strategy for this question.

Code: Trigger Function

```
CREATE OR REPLACE FUNCTION graduation_date()
RETURNS TRIGGER AS $$ 
BEGIN
    IF NEW.graduate > CURRENT_DATE
    THEN
        NEW.graduate := CURRENT_DATE;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Here, the trigger can be combined because both `INSERT` and `UPDATE` have access to the `NEW` variable.

Code: Trigger

```
CREATE TRIGGER no_future_graduation
BEFORE INSERT OR UPDATE ON student
FOR EACH ROW EXECUTE FUNCTION graduation_date();
```

Although we have already not used `RETURN NULL` to avoid rejecting the insert/update, the operation may still fail. This happens when `CURRENT_DATE < year`.

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] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. 2nd. USA: McGraw-Hill, Inc., 2000. ISBN: 0072440422.
- [4] W3schools Online Web Tutorials. <https://www.w3schools.com/>. [Online; last accessed 2025].