PostgreSQL audit logging using triggers

vladmihalcea.com/postgresql-audit-logging-triggers/

vladmihalcea

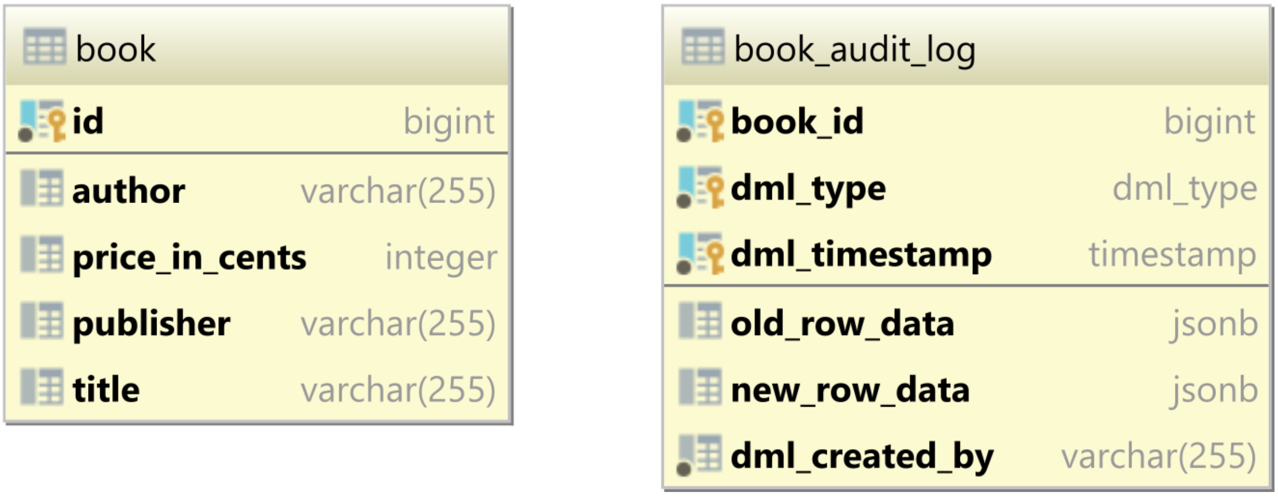
Introduction

In this article, we are going to see how we can implement an audit logging mechanism using PostgreSQL database triggers to store the CDC (Change Data Capture) records.

Thanks to JSON column types, we can store the row state in a single column, therefore not needing to add a new column in the audit log table every time a new column is being added to the source database table.

Database tables

Let’s assume we are developing a library application that uses the following two tables:



book	
id	bigint
author	varchar(255)
price_in_cents	integer
publisher	varchar(255)
title	varchar(255)

book_audit_log	
book_id	bigint
dml_type	dml_type
dml_timestamp	timestamp
old_row_data	jsonb
new_row_data	jsonb
dml_created_by	varchar(255)

In the `book` table, we are going to store all the books offered by our library, and the `book_audit_log` table is going to store the CDC (Change Data Capture) events that happened whenever an INSERT, UPDATE, or DELETE DML statement got executed on the `book` table.

The `book_audit_log` table is created like this:

```
CREATE TABLE IF NOT EXISTS book_audit_log (
 book_id bigint NOT NULL,
 old_row_data jsonb,
 new_row_data jsonb,
 dml_type dml_type NOT NULL,
 dml_timestamp timestamp NOT NULL,
 dml_created_by varchar(255) NOT NULL,
 PRIMARY KEY (book_id, dml_type, dml_timestamp)
)
```

The `book_id` column stores the identifier of the associated `book` table record that was inserted, updated, or deleted by the current executing DML statement.

The `old_row_data` is a JSONB column that captures the state of the `book` row before the execution of the current INSERT, UPDATE, or DELETE statement.

The `new_row_data` is a JSONB column that will capture the state of the `book` row after the execution of the current INSERT, UPDATE, or DELETE statement.

The `dml_type` column stores the type of the current executing DML statement (e.g., INSERT, UPDATE, and DELETE). The `dml_type` type is a PostgreSQL enumeration type, that was created like this:

```
CREATE TYPE dml_type AS ENUM ('INSERT', 'UPDATE', 'DELETE')
```

The `dml_timestamp` column stores the current timestamp.

The `dml_created_by` column stores the application user who generated the current INSERT, UPDATE, or DELETE DML statement.

The Primary Key of the `book_audit_log` is a composite of the `book_id` , `dml_type` , and `dml_timestamp` since a `book` record can have multiple associated `book_audit_log` records.

PostgreSQL audit logging triggers

To capture the INSERT, UPDATE, and DELETE DML statements on the `book` table, we need to create a trigger function that looks as follows:

```
CREATE OR REPLACE FUNCTION book_audit_trigger_func()
RETURNS trigger AS $body$
BEGIN
 if (TG_OP = 'INSERT') then
 INSERT INTO book_audit_log (
 book_id,
 old_row_data,
 new_row_data,
 dml_type,
 dml_timestamp,
 dml_created_by
 )
 VALUES(
 NEW.id,
 null,
 to_jsonb(NEW),
 'INSERT',
 CURRENT_TIMESTAMP,
 current_setting('var.logged_user')
 );

 RETURN NEW;
 elsif (TG_OP = 'UPDATE') then
 INSERT INTO book_audit_log (
 book_id,
 old_row_data,
 new_row_data,
 dml_type,
 dml_timestamp,
 dml_created_by
 )
 VALUES(
 NEW.id,
 to_jsonb(OLD),
 to_jsonb(NEW),
 'UPDATE',
 CURRENT_TIMESTAMP,
 current_setting('var.logged_user')
 );

 RETURN NEW;
 elsif (TG_OP = 'DELETE') then
 INSERT INTO book_audit_log (
 book_id,
 old_row_data,
 new_row_data,
 dml_type,
 dml_timestamp,
 dml_created_by
 )
 VALUES(
 OLD.id,
 to_jsonb(OLD),
 null,
 'DELETE',
 CURRENT_TIMESTAMP,
 current_setting('var.logged_user')
 );
 RETURN OLD;
 end if;

END;
$body$
LANGUAGE plpgsql
```

In order for the `book_audit_trigger_func` function to be executed after a `book` table record is inserted, updated or deleted, we have to define the following trigger:

```
CREATE TRIGGER book_audit_trigger
AFTER INSERT OR UPDATE OR DELETE ON book
FOR EACH ROW EXECUTE FUNCTION book_audit_trigger_func()
```

The `book_audit_trigger_func` function can be explained as follows:

- the `TG_OP` variable provides the type of the current executing DML statement.
- the `NEW` keyword is also a special variable that stores the state of the current modifying record after the current DML statement is executed.
- the `OLD` keyword is also a special variable that stores the state of the current modifying record before the current DML statement is executed.
- the `to_jsonb` PostgreSQL function allows us to transform a table row to a JSONB object, that's going to be saved in the `old_row_data` or `new_row_data` table columns.
- the `dml_timestamp` value is set to the `CURRENT_TIMESTAMP`
- the `dml_created_by` column is set to the value of the `var.logged_user` PostgreSQL session variable, which was previously set by the application with the currently logged user, like this:

```
Session session = entityManager.unwrap(Session.class);
Dialect dialect = session.getSessionFactory()
 .unwrap(SessionFactoryImplementor.class)
 .getJdbcServices()
 .getDialect();
session.doWork(connection -> {
 update(
 connection,
 String.format(
 "SET LOCAL var.logged_user = '%s'",
 ReflectionUtils.invokeMethod(
 dialect,
 "escapeLiteral",
 LoggedUser.get()
 )
 )
 );
});
```

> Notice that we used `SET LOCAL` as we want the variable to be removed after the current transaction is committed or rolled back. This is especially useful when using connection pooling.

Testing time

When executing an INSERT statement on the `book` table:

```
INSERT INTO book (
 id,
 author,
 price_in_cents,
 publisher,
 title
)
VALUES (
 1,
 'Vlad Mihalcea',
 3990,
 'Amazon',
 'High-Performance Java Persistence 1st edition'
)
```

We can see that a record is inserted in the `book_audit_log` that captures the INSERT statement that was just executed on the `book` table:

```
| book_id | old_row_data | new_row_data | dml_type | dml_timestamp |
dml_created_by |
|---------|--------------|-------------------------------------------------------
---------------------------------------------------------------------------------
-----|----------|----------------------------|----------------|
| 1 | | {"id": 1, "title": "High-Performance Java Persistence 1st edition",
"author": "Vlad Mihalcea", "publisher": "Amazon", "price_in_cents": 3990} | INSERT
| 2020-08-25 13:19:57.073026 | Vlad Mihalcea |
```

When updating the `book` table row:

```
UPDATE book
SET price_in_cents = 4499
WHERE id = 1
```

We can see that a new record is going to be added to the `book_audit_log` by the `book_audit_trigger`:

```
| book_id | old_row_data | new_row_data | dml_type | dml_timestamp |
dml_created_by |
|---------|-------------------------------------------------------------------------
-------------------------------------------------------------------------|---------
---------------------------------------------------------------------------------
---------------------------------------------------|----------|------------------
----------|----------------|
| 1 | | {"id": 1, "title": "High-Performance Java Persistence 1st edition",
"author": "Vlad Mihalcea", "publisher": "Amazon", "price_in_cents": 3990} | INSERT
| 2020-08-25 13:19:57.073026 | Vlad Mihalcea |
| 1 | {"id": 1, "title": "High-Performance Java Persistence 1st edition",
"author": "Vlad Mihalcea", "publisher": "Amazon", "price_in_cents": 3990} | {"id":
1, "title": "High-Performance Java Persistence 1st edition", "author": "Vlad
Mihalcea", "publisher": "Amazon", "price_in_cents": 4499} | UPDATE | 2020-08-25
13:21:15.006365 | Vlad Mihalcea |
```

When deleting the `book` table row:

```
DELETE FROM book
WHERE id = 1
```

A new record is added to the `book_audit_log` by the `book_audit_trigger` :

```
| book_id | old_row_data | new_row_data | dml_type | dml_timestamp |
dml_created_by |
|---------|-------------------------------------------------------------------
-----------------------------------------------------------------------|---------
---------------------------------------------------------------------------------
----------------------------------------------------|----------|------------------
----------|----------------|
| 1 | | {"id": 1, "title": "High-Performance Java Persistence 1st edition",
"author": "Vlad Mihalcea", "publisher": "Amazon", "price_in_cents": 3990} | INSERT
| 2020-08-25 13:19:57.073026 | Vlad Mihalcea |
| 1 | {"id": 1, "title": "High-Performance Java Persistence 1st edition",
"author": "Vlad Mihalcea", "publisher": "Amazon", "price_in_cents": 3990} | {"id":
1, "title": "High-Performance Java Persistence 1st edition", "author": "Vlad
Mihalcea", "publisher": "Amazon", "price_in_cents": 4499} | UPDATE | 2020-08-25
13:21:15.006365 | Vlad Mihalcea |
| 1 | {"id": 1, "title": "High-Performance Java Persistence 1st edition",
"author": "Vlad Mihalcea", "publisher": "Amazon", "price_in_cents": 4499} | |
DELETE | 2020-08-25 13:21:58.499881 | Vlad Mihalcea |
```

Awesome, right?

Conclusion

There are many ways to implement an audit logging mechanism. If you are using Hibernate, a very simple solution is to use Hibernate Envers.

If you are not using Hibernate or if you want to capture the CDC events no matter how the DML statements are generated, then a database trigger solution, such as the one presented in this article, is quite straightforward to implement. Storing the old and new row states in JSON columns is a very good idea since it allows us to reuse the same function even if the source table structure changes.

Another option is to use a dedicated CDC framework, like Debezium, which extracts the CDC events from the PostgreSQL WAL (Write-Ahead Log). This solution can be very efficient since it works asynchronously, so it has no impact on the current executing OLTP transactions. However, setting up Debezium and running it in production is going to be much more challenging since Debezium requires Apache Kafka and ZooKeeper as well.