SQLite Trigger

Summary: this tutorial discusses SQLite trigger, which is a database object fired automatically when the data in a table is changed.

What is an SQLite trigger

An SQLite trigger is a named database object that is executed automatically when an `INSERT`, `UPDATE` or `DELETE` statement is issued against the associated table.

When do we need SQLite triggers

You often use triggers to enable sophisticated auditing. For example, you want to log the changes in the sensitive data such as salary and address whenever it changes.

In addition, you use triggers to enforce complex business rules centrally at the database level and prevent invalid transactions.

SQLite CREATE TRIGGER statement

To create a new trigger in SQLite, you use the `CREATE TRIGGER` statement as follows:

```
CREATE TRIGGER [IF NOT EXISTS] trigger_name
   [BEFORE|AFTER|INSTEAD OF] [INSERT|UPDATE|DELETE]
   ON table_name
   [WHEN condition]
BEGIN
 statements;
END;
```

In this syntax:

- First, specify the name of the trigger after the `CREATE TRIGGER` keywords.
- Next, determine when the trigger is fired such as `BEFORE`, `AFTER`, or `INSTEAD OF`. You can create `BEFORE` and `AFTER` triggers on a table. However, you can only create an `INSTEAD OF` trigger on a view.
- Then, specify the event that causes the trigger to be invoked such as `INSERT`, `UPDATE`, or `DELETE`.
- After that, indicate the table to which the trigger belongs.
- Finally, place the trigger logic in the `BEGIN END` block, which can be any valid SQL statements.

If you combine the time when the trigger is fired and the event that causes the trigger to be fired, you have a total of 9 possibilities:

- `BEFORE INSERT`
- `AFTER INSERT`
- `BEFORE UPDATE`
- `AFTER UPDATE`
- `BEFORE DELETE`
- `AFTER DELETE`
- `INSTEAD OF INSERT`
- `INSTEAD OF DELETE`
- `INSTEAD OF UPDATE`

Suppose you use a `UPDATE` statement to update 10 rows in a table, the trigger that associated with the table is fired 10 times. This trigger is called `FOR EACH ROW` trigger. If the trigger associated with the table is fired one time, we call this trigger a `FOR EACH STATEMENT` trigger.

As of version 3.9.2, SQLite only supports `FOR EACH ROW` triggers. It has not yet supported the `FOR EACH STATEMENT` triggers.

If you use a condition in the `WHEN` clause, the trigger is only invoked when the condition is true. In case you omit the `WHEN` clause, the trigger is executed for all rows.

Notice that if you drop a table, all associated triggers are also deleted. However, if the trigger references other tables, the trigger is not removed or changed if other tables are removed or updated.

For example, a trigger references to a table named `people`, you drop the `people` table or rename it, you need to manually change the definition of the trigger.

You can access the data of the row being inserted, deleted, or updated using the `OLD` and `NEW` references in the form: `OLD.column_name` and `NEW.column_name`.

the `OLD` and `NEW` references are available depending on the event that causes the trigger to be fired.

The following table illustrates the rules.:

Action	Reference
INSERT	NEW is available
UPDATE	Both NEW and OLD are available
DELETE	OLD is available

SQLite triggers examples

Let's create a new table called leads to store all business leads of the company.

```
CREATE TABLE leads (
	id integer PRIMARY KEY,
	first_name text NOT NULL,
	last_name text NOT NULL,
	phone text NOT NULL,
	email text NOT NULL,
	source text NOT NULL
);
```

1) SQLite `BEFORE INSERT` trigger example

Suppose you want to validate the email address before inserting a new lead into the `leads` table. In this case, you can use a `BEFORE INSERT` trigger.

First, create a `BEFORE INSERT` trigger as follows:

```
CREATE TRIGGER validate_email_before_insert_leads
   BEFORE INSERT ON leads
BEGIN
   SELECT
      CASE
	WHEN NEW.email NOT LIKE '%_@__%.__%' THEN
   	  RAISE (ABORT,'Invalid email address')
       END;
END;
```

We used the `NEW` reference to access the email column of the row that is being inserted.

To validate the email, we used the `LIKE` operator to determine whether the email is valid or not based on the email pattern. If the email is not valid, the `RAISE` function aborts the insert and issues an error message.

Second, insert a row with an invalid email into the `leads` table.

```
INSERT INTO leads (first_name,last_name,email,phone)
VALUES('John','Doe','jjj','4089009334');
```

SQLite issued an error: "Invalid email address" and aborted the execution of the insert.

Third, insert a row with a valid email.

```
INSERT INTO leads (first_name, last_name, email, phone)
VALUES ('John', 'Doe', 'john.doe@sqlitetutorial.net', '4089009334');
```

Because the email is valid, the insert statement executed successfully.

```
SELECT
	first_name,
	last_name,
	email,
	phone
FROM
	leads;
```

first_name	last_name	email	phone
John	Doe	john.doe@sqlitetutorial.net	4089009334

2) SQLite `AFTER UPDATE` trigger example

The phones and emails of the leads are so important that you can't afford to lose this information. For example, someone accidentally updates the email or phone to the wrong ones or even delete it.

To protect this valuable data, you use a trigger to log all changes which are made to the phone and email.

First, create a new table called `lead_logs` to store the historical data.

```
CREATE TABLE lead_logs (
	id INTEGER PRIMARY KEY,
	old_id int,
	new_id int,
	old_phone text,
	new_phone text,
	old_email text,
	new_email text,
	user_action text,
	created_at text
);
```

Second, create an `AFTER UPDATE` trigger to log data to the `lead_logs` table whenever there is an update in the `email` or `phone` column.

```
CREATE TRIGGER log_contact_after_update
   AFTER UPDATE ON leads
   WHEN old.phone <> new.phone
        OR old.email <> new.email
BEGIN
	INSERT INTO lead_logs (
		old_id,
		new_id,
		old_phone,
		new_phone,
		old_email,
		new_email,
		user_action,
		created_at
	)
VALUES
	(
		old.id,
		new.id,
		old.phone,
		new.phone,
		old.email,
		new.email,
		'UPDATE',
		DATETIME('NOW')
	) ;
END;
```

You notice that in the condition in the `WHEN` clause specifies that the trigger is invoked only when there is a change in either email or phone column.

Third, update the last name of `John` from `Doe` to `Smith`.

```
UPDATE leads
SET
   last_name = 'Smith'
WHERE
   id = 1;
```

The trigger `log_contact_after_update` was not invoked because there was no change in email or phone.

Fourth, update both email and phone of `John` to the new ones.

```
UPDATE leads
SET
   phone = '4089998888',
   email = 'john.smith@sqlitetutorial.net'
WHERE
   id = 1;
```

If you check the log table, you will see there is a new entry there.

```
SELECT
   old_phone,
   new_phone,
   old_email,
   new_email,
   user_action
FROM
   lead_logs;
```

old_phone	new_phone	old_email	new_email	user_action
4089009334	4089998888	john.doe@sqlitetutorial.	john.smith@sqlitetutorial.net	UPDATE

You can develop the `AFTER INSERT` and `AFTER DELETE` triggers to log the data in the `lead_logs` table as an excercise.

SQLite DROP TRIGGER statement

To drop an existing trigger, you use the `DROP TRIGGER` statement as follows:

```
DROP TRIGGER [IF EXISTS] trigger_name;
```

In this syntax:

- First, specify the name of the trigger that you want to drop after the `DROP TRIGGER` keywords.
- Second, use the `IF EXISTS` option to delete the trigger only if it exists.

Note that if you drop a table, SQLite will automatically drop all triggers associated with the table.

For example, to remove the `validate_email_before_insert_leads` trigger, you use the following statement:

```
DROP TRIGGER validate_email_before_insert_leads;
```

In this tutorial, we have introduced you to SQLite triggers and show you how to create and drop triggers from the database.

Was this tutorial helpful ?

PREVIOUSLY
SQLite Expression-based Index

UP NEXT
SQLite INSTEAD OF Triggers

Search ...

GETTING STARTED

- What Is SQLite
- Download & Install SQLite
- SQLite Sample Database
- SQLite Commands

SQLITE TUTORIAL

- SQLite Select
- SQLite Order By
- SQLite Select Distinct
- SQLite Where
- SQLite Limit
- SQLite BETWEEN
- SQLite IN
- SQLite Like
- SQLite IS NULL
- SQLite GLOB
- SQLite Join
- SQLite Inner Join
- SQLite Left Join
- SQLite Cross Join
- SQLite Self-Join
- SQLite Full Outer Join
- SQLite Group By
- SQLite Having
- SQLite Union
- SQLite Except
- SQLite Intersect
- SQLite Subquery
- SQLite EXISTS
- SQLite Case
- SQLite Insert
- SQLite Update
- SQLite Delete
- SQLite Replace
- SQLite Transaction

SQLITE DATA DEFINITION

- SQLite Data Types
- SQLite Date & Time
- SQLite Create Table
- SQLite Primary Key
- SQLite Foreign Key
- SQLite NOT NULL Constraint
- SQLite UNIQUE Constraint
- SQLite CHECK constraints
- SQLite AUTOINCREMENT
- SQLite Alter Table
- SQLite Rename Column
- SQLite Drop Table
- SQLite Create View
- SQLite Drop View
- SQLite Index
- SQLite Expression-based Index
- SQLite Trigger
- SQLite VACUUM
- SQLite Transaction
- SQLite Full-text Search

SQLITE TOOLS

- SQLite Commands
- SQLite Show Tables
- SQLite Describe Table
- SQLite Dump
- SQLite Import CSV
- SQLite Export CSV

SQLITE FUNCTIONS

- SQLite AVG
- SQLite COUNT
- SQLite MAX
- SQLite MIN
- SQLite SUM

SQLITE INTERFACES

- SQLite PHP
- SQLite Node.js
- SQLite Java
- SQLite Python

LATEST TUTORIALS

- SQLite IIF
- SQLite Generated Columns
- SQLite Getting Started
- SQLite Programming Interfaces
- SQLite Concat
- SQLite INSTEAD OF Triggers
- SQLite Join
- SQLite IS NULL
- SQLite Rename Column
- SQLite DROP VIEW

SITE LINKS

- Home
- About
- Contact
- Resources
- Privacy Policy