Лабораторная работа №17
Создание и удаление триггеров в СУБД MySQL.
Цель: Сформировать умения создавать и удалять триггеры, реагирующие на действия пользователей.

Задание.

1. Изучить виды триггеров, команды создания триггеров используя краткие теоретические сведения и порядок выполнения работы.
2. Используя **таблицы из индивидуального задания см. л.р. 15** (или из учебной БД **world**), создать триггеры:
   2.1 Создать триггер BEFOR INSERT для проверки вводимых значений данных.
   2.2 Создать триггер AFTER INSERT после успешного добавления данных.
   2.3 Создать триггер BEFOR UPDATE для проверки обновляемых значений данных.
   2.4 Создать триггер AFTER UPDATE для фиксирования даты и времени обновления, старого значения поля, нового значения поля, логина пользователя, производившего изменения в таблице логирования.
   2.5 Создать триггер BEFORE DELETE для обеспечения целостности ссылок в разных связанных таблицах.
   2.6 Создать триггер BEFORE DELETE для регистрации факта удаления записи из таблицы в таблице логирования.
   2.7 Удалить триггер (любой на выбор).

Шаг 1 — Создание тестовой базы данных

```
1. Create database test_db;
2.
```

```
Output

Query OK, 1 row affected (0.00 sec)
```

Далее переходите к `test_db` с помощью:

```
1. Use test_db;
2.
```

```
Output
```

```
Database changed
```

Начинайте с создания таблицы `customers`. В этой таблице будут храниться записи клиентов, включая `customer_id`, `customer_name` и `level`. Будет два типа клиентов: `BASIC` и `VIP`.

```
1. Create table customers(customer_id BIGINT PRIMARY KEY, customer_name
   VARCHAR(50), level VARCHAR(50) ) ENGINE=INNODB;
2.
```

```
Output

Query OK, 0 rows affected (0.01 sec)
```

Теперь, добавьте несколько записей в таблицу `customers`. Для этого выполните следующие команды одну за другой:

```
1. Insert into customers (customer_id, customer_name, level )values('1','JOHN
   DOE','BASIC');
2.
3. Insert into customers (customer_id, customer_name, level )values('2','MARY
   ROE','BASIC');
4.
5. Insert into customers (customer_id, customer_name, level )values('3','JOHN
   DOE','VIP');
6.
```

После выполнения каждой команды `INSERT` вы увидите следующий вывод:

```
Output

Query OK, 1 row affected (0.01 sec)
```

Чтобы убедиться, что тестовые записи были успешно вставлены, выполните команду `SELECT`:

```
1. Select * from customers;
2.
```

```
Output

+-------------+---------------+-------+

| customer_id | customer_name | level |

+-------------+---------------+-------+

|           1 | JOHN DOE      | BASIC |

|           2 | MARY ROE      | BASIC |

|           3 | JOHN DOE      | VIP   |

+-------------+---------------+-------+

3 rows in set (0.00 sec)
```

Затем создайте другую таблицу `customers` для хранения соответствующей информации об учетной записи клиентов. Таблица будет содержать поля `customer_id` и `status_notes`.

Запустите следующую команду:

```
Create table customer_status(customer_id BIGINT PRIMARY KEY, status_notes VARCHAR(50)) ENGINE=INNODB;
```

Далее создайте таблицу `sales`. В этой таблице будут храниться данные о продажах, имеющих отношение к разным клиентам в столбце `customer_id`:

```
Create table sales(sales_id BIGINT PRIMARY KEY, customer_id BIGINT, sales_amount DOUBLE ) ENGINE=INNODB;
```

```
Output

Query OK, 0 rows affected (0.01 sec)
```

Вы сможете добавить тестовые данные в колонку `sales` на следующих этапах во время тестирования триггеров. Далее создайте таблицу `audit_log` для регистрации обновлений, внесенных в таблицу `sales` при имплементации триггера `AFTER UPDATE` в шаге 5:

```
Create table audit_log(log_id BIGINT PRIMARY KEY AUTO_INCREMENT, sales_id BIGINT, previous_amount DOUBLE, new_amount DOUBLE, updated_by VARCHAR(50), updated_on DATETIME ) ENGINE=INNODB;
```

```
Output

Query OK, 0 rows affected (0.02 sec)
```

Имея базу данных `test_db` и четыре таблицы, теперь вы можете перейти к работе с различными триггерами MySQL в вашей базе данных.

Шаг 2 — Создание триггера Before Insert

На этом этапе вы изучите синтаксис триггера MySQL перед тем, как применить эту логику для создания триггера `BEFORE INSERT`, который проверяет поле `sales_amount` перед вставкой данных в таблицу `sales`.

Общий синтаксис для создания триггера MySQL показан в следующем примере:

```
DELIMITER //

CREATE TRIGGER [TRIGGER_NAME]

[TRIGGER TIME] [TRIGGER EVENT]

ON [TABLE]

FOR EACH ROW
```

```
[TRIGGER BODY]//

DELIMITER ;
```

Структура триггера включает:

`DELIMITER //`: разделитель MySQL по умолчанию — это `;`. Его нужно заменить на что-то другое, для того, чтобы MySQL рассматривал следующие строки, как одну команду, пока не достигнет пользовательского разделителя. В данном примере в качестве разделителя используется `//`, а стандартный разделитель `;` стоит в конце.

`[TRIGGER_NAME]`: триггер должен иметь имя, и вы можете указать его именно здесь.

`[TRIGGER TIME]`: триггер может быть вызван в разные моменты времени. MySQL позволяет определить, когда запускать триггер — до или после операции с базой данных.

`[TRIGGER EVENT]`: триггеры могут быть вызваны только операциями `INSERT`, `UPDATE` и `DELETE`. Вы можете использовать любое из значений в зависимости от того, чего вы хотите достичь.

`[TABLE]`: любой триггер, который вы создаете в своей базе данных MySQL, должен быть связан с таблицей.

`FOR EACH ROW`: этот оператор позволяет MySQL выполнять код триггера для каждой строки, на которую влияет триггер.

`[TRIGGER BODY]`: код, который выполняется при вызове триггера, называется *trigger body*. Это может быть один SQL-оператор или несколько команд. Обратите внимание, если вы выполняете несколько SQL-операторов в теле триггера, вы должны заключить их в блок `BEGIN... END`.

Примечание: при создании тела триггера вы можете использовать ключевые слова `OLD` и `NEW` для доступа к старым и новым значениям колонки, введенным во время операции `INSERT`, `UPDATE` и `DELETE`. В триггере `DELETE` может быть использовано только ключевое слово `OLD` (подробнее об этом в шаге 4).

Теперь вы можете создать свой первый триггер `BEFORE INSERT`. Триггер будет связан с таблицей `sales` и будет вызываться перед вставкой записи для проверки `sales_amount`. Функция триггера состоит в том, чтобы проверить, превышает ли значение `sales_amount`, вставляемое в таблицу продаж, величину `10000`, и выдать ошибку, если это так.

Убедитесь, что вы вошли на сервер MySQL. Затем введите следующие команды MySQL одну за другой:

```
1. DELIMITER //
2.
3. CREATE TRIGGER validate_sales_amount
4.
5. BEFORE INSERT
6.
7. ON sales
```

```
8.
9. FOR EACH ROW
10.
11.   IF NEW.sales_amount>10000 THEN
12.
13.   SIGNAL SQLSTATE '45000'
14.
15.   SET MESSAGE_TEXT = 'Sale has exceeded the allowed amount of 10000.';
16.
17.   END IF//
18.
19.   DELIMITER ;
20.
```

Используйте `IF... THEN... END IF` для оценки того, находится ли сумма, указанная в операторе `INSERT`, в пределах вашего диапазона. Триггер может извлечь новое значение `sales_amount`, используя ключевое слово `NEW`.

Чтобы вызвать общее сообщение об ошибке, используются следующие строки для информирования пользователя:

```
SIGNAL SQLSTATE '45000'

SET MESSAGE_TEXT = 'Sale has exceeded the allowed amount of 10000.';
```

Далее вставьте запись `sales_amount` со значением `11000` в таблицу `sales`, чтобы проверить, остановит ли триггер операцию:

```
1. Insert into sales(sales_id, customer_id, sales_amount)
   values('1','1','11000');
2.
```

```
Output

ERROR 1644 (45000): Sale has exceeded the allowed amount of 10000.
```

Эта ошибка показывает, что код триггера работает должным образом.

Теперь попробуйте новую запись со значением `7500`, чтобы проверить правильность действия команды:

```
1. Insert into  sales(sales_id, customer_id, sales_amount)
   values('1','1','7500');
2.
```

Поскольку значение находится в рекомендованном диапазоне, вы увидите следующий вывод:

```
Output

Query OK, 1 row affected (0.01 sec)
```

Для подтверждения вставки данных запустите следующую команду:

```
1. Select * from sales;
2.
```

Вывод подтверждает вставку данных в таблицу:

```
Output

+----------+-------------+--------------+

| sales_id | customer_id | sales_amount |

+----------+-------------+--------------+

|        1 |           1 |         7500 |

+----------+-------------+--------------+

1 row in set (0.00 sec)
```

На этом этапе вы протестировали способность триггеров проверять данные перед вставкой в базу данных.

Теперь поработайте с триггером `AFTER INSERT` для сохранения связанной информации в разных таблицах.

Шаг 3 — Создание триггера After Insert

Триггеры `AFTER INSERT` выполняются после успешной вставки записей в таблицу. Эта функция может использоваться для автоматического запуска других бизнес-логик. Например, в банковских приложениях триггер `AFTER INSERT` может закрывать кредитный счет, когда клиент завершает выплату кредита. Триггер может отслеживать все платежи, внесенные в таблицу транзакций, и автоматически закрывать кредит, как только кредитный баланс будет равен нулю.

На этом этапе вы поработаете с таблицей `customer_status`, используя триггер `AFTER INSERT` для ввода связанных клиентских записей.

Для создания триггера `AFTER INSERT` введите следующие команды:

```
1. DELIMITER //
2.
3. CREATE TRIGGER customer_status_records
4.
5. AFTER INSERT
6.
7. ON customers
8.
9. FOR EACH ROW
10.
11.   Insert into customer_status(customer_id, status_notes)
   VALUES(NEW.customer_id, 'ACCOUNT OPENED SUCCESSFULLY')//
12.
13.   DELIMITER ;
14.
```

```
Output

Query OK, 0 rows affected (0.00 sec)
```

Таким образом вы инструктируете MySQL сохранить еще одну запись в таблицу `customer_status`, как только происходит вставка новой клиентской записи в таблицу `customers`.

Теперь вставьте новую запись в таблицу `customers`, чтобы убедиться, что код триггера вызывается:

```
1. Insert into customers (customer_id, customer_name, level
   )values('4','DAVID DOE','VIP');
2.
```

```
Output

Query OK, 1 row affected (0.01 sec)
```

После успешной вставки записи убедитесь, что запись нового статуса была добавлена в таблицу `customer_status`:

```
1. Select * from customer_status;
2.
```

```
Output

+-------------+-----------------------------+

| customer_id | status_notes                |

+-------------+-----------------------------+

|           4 | ACCOUNT OPENED SUCCESSFULLY |

+-------------+-----------------------------+

1 row in set (0.00 sec)
```

Вывод подтверждает успешную работу триггера.

Триггер `AFTER INSERT` полезен для мониторинга жизненного цикла клиента. В производственной среде учетные записи клиентов могут проходить различные этапы, например открытие, приостановка и закрытие счета.

На следующем этапе вы будете работать с триггерами `UPDATE`.

Шаг 4 — Создание триггера Before Update

Триггер `BEFORE UPDATE` схож с триггером `BEFORE INSERT`, разница заключается в том, когда они вызываются. Вы можете использовать триггер `BEFORE UPDATE` для проверки бизнес-логики перед обновлением записи. Для проверки используйте таблицу `customers`, в которую вы уже вставили некоторые данные.

В базе данных есть два типа клиентов. В этом примере после того, как учетная запись клиента будет обновлена до уровня `VIP`, она не сможет быть понижена до уровня `BASIC`. Чтобы применить такое правило, создайте триггер `BEFORE UPDATE`, который будет выполняться перед оператором `UPDATE`, как показано ниже. Если

пользователь базы данных попытается понизить клиента до уровня `BASIC` с уровня `VIP`, будет активировано определяемое пользователем исключение.

Введите следующие команды SQL одну за другой, чтобы создать триггер `BEFORE UPDATE`:

```
DELIMITER //

CREATE TRIGGER validate_customer_level

BEFORE UPDATE

ON customers

FOR EACH ROW

  IF OLD.level='VIP' THEN

  SIGNAL SQLSTATE '45000'

  SET MESSAGE_TEXT = 'A VIP customer can not be downgraded.';

  END IF //

  DELIMITER ;

```

Copy

Используйте ключевое слово `OLD` для фиксации уровня, предоставленного пользователем при выполнении команды `UPDATE`. Опять же, вы используете `IF... THEN... END IF`, чтобы сообщить пользователю об общей ошибке.

Далее выполните следующую SQL команду, которая попытается понизить учетную запись клиента, имеющую идентификатор `customer_id`, равный `3`:

```
Update customers set level='BASIC' where customer_id='3';

```

Вы увидите следующий вывод, предоставляющий `SET MESSAGE_TEXT`:

```
Output

ERROR 1644 (45000): A VIP customer can not be downgraded.
```

Если вы выполните ту же команду для клиента уровня `BASIC` и попытаетесь повысить учетную запись до уровня `VIP`, команда выполнится успешно:

```
Update customers set level='VIP' where customer_id='1';

```

```
Output

Rows matched: 1  Changed: 1  Warnings: 0
```

Вы использовали триггер `BEFORE UPDATE` для применения бизнес-правила. Теперь перейдем к использованию триггера `AFTER UPDATE` для ведения журнала аудита.

Шаг 5 — Создание триггера After Update

Триггер `AFTER UPDATE` вызывается после успешного обновления записи в базе данных. Такое поведение триггера подходит для ведения журнала аудита. В многопользовательской среде администратор с целью аудита может просмотреть историю пользователей, обновляющих записи в конкретной таблице.

Вы создаете триггер, который регистрирует активность обновления таблицы `sales`. Наша таблица `audit_log` будет содержать информацию о пользователях MySQL, обновляющих таблицу `sales`, дату обновления `date`, а также новые `new` и старые `old` значения `sales_amount`.

Для создания триггера, выполните следующие команды SQL:

```
1. DELIMITER //
2.
3. CREATE TRIGGER log_sales_updates
4.
5. AFTER UPDATE
6.
7. ON sales
8.
9. FOR EACH ROW
10.
11.   Insert into audit_log(sales_id, previous_amount, new_amount,
   updated_by, updated_on) VALUES (NEW.sales_id,OLD.sales_amount,
   NEW.sales_amount,(SELECT USER()), NOW() )//
12.
13.   DELIMITER ;
14.
```

Вы вставляете новую запись в таблицу `audit_log`. Вы используете ключевое слово `NEW` для получения значения `sales_id` и нового значения `sales_amount`. Также вы используете ключевое слово `OLD` для получения предыдущего значения `sales_amount`, если вы хотите зарегистрировать обе суммы для аудита.

Команда `SELECT USER()` извлекает текущего пользователя, выполняющего операцию, а оператор `NOW()` извлекает значение текущей даты и времени с сервера MySQL.

Теперь, если пользователь попытается обновить значение какой-либо записи в таблице `sales`, триггер `log_sales_updates` вставит новую запись в таблицу `audit_log`.

Давайте создадим новую запись о продажах со случайным значением `sales_id`, равным `5`, и попробуем обновить ее. Сначала вставьте запись о продажах:

```
1. Insert into sales(sales_id, customer_id, sales_amount) values('5',
   '2','8000');
2.
```

```
Output

Query OK, 1 row affected (0.00 sec)
```

Затем обновите запись:

```
1. Update sales set sales_amount='9000' where sales_id='5';
2.
```

Вывод должен выглядеть так:

```
Output

Rows matched: 1  Changed: 1  Warnings: 0
```

Теперь выполните следующую команду, чтобы проверить, смог ли триггер `AFTER UPDATE` зарегистрировать новую запись в таблице `audit_log`:

```
1. Select * from audit_log;
2.
```

Триггер зарегистрировал обновление. Ваш вывод должен показать предыдущую сумму `sales_amount` и новую сумму `new amount`, зарегистрированную пользователем, который обновил запись:

```
Output

+--------+----------+-----------------+------------+----------------+-----
----------------+

| log_id | sales_id | previous_amount | new_amount | updated_by     |
updated_on          |

+--------+----------+-----------------+------------+----------------+-----
----------------+

|      1 |        5 |            8000 |       9000 | root@localhost |
2019-11-07 09:28:36 |

+--------+----------+-----------------+------------+----------------+-----
----------------+

1 row in set (0.00 sec)
```

Также в таблице вы увидите дату и время, когда было выполнено обновление, что важно для аудита.

Далее вы будете использовать триггер `DELETE` для обеспечения целостности ссылок на уровне базы данных.

Шаг 2 — Создание триггера Before Delete

Триггеры `BEFORE DELETE` вызываются до выполнения операции `DELETE` в таблице. Этот вид триггеров обычно используется для обеспечения целостности ссылок в разных связанных таблицах. Например, каждая запись в таблице `sales` связана с

записью `customer_id` из таблицы `customers`. Если пользователь базы данных удалил из таблицы `customers` запись, у которой есть связанная запись в таблице `sales`, у вас не будет возможности узнать, какой клиент был связан с этой записью.

Избежать подобных ситуаций и сделать логику более надежной позволит создание триггера `BEFORE DELETE`. Выполните следующие SQL команды одну за другой:

```
DELIMITER //

CREATE TRIGGER validate_related_records

BEFORE DELETE

ON customers

FOR EACH ROW

   IF OLD.customer_id in (select customer_id from sales) THEN

   SIGNAL SQLSTATE '45000'

   SET MESSAGE_TEXT = 'The customer has a related sales record.';

   END IF//

   DELIMITER ;

```

Теперь попробуйте удалить клиента, у которого есть связанная запись в таблице sales:

```
Delete from customers where customer_id='2';

```

В результате вы получите следующий вывод:

```
Output

ERROR 1644 (45000): The customer has a related sales record.
```

Триггер `BEFORE DELETE` может предотвратить случайное удаление связанной информации в базе данных.

В некоторых ситуациях может потребоваться удалить из разных связанных таблиц все записи, связанные с конкретной записью. В этой ситуации возможно использовать триггер `AFTER DELETE`, который вы протестируете в следующем шаге.

Шаг 5 — Создание триггера After Delete

Триггеры `AFTER DELETE` активируются, когда запись была успешно удалена. Примером использования триггера `AFTER DELETE` является ситуация, когда скидка, которую получает конкретный клиент, определяется количеством покупок, совершенных этим клиентом в течение определенного периода. Если некоторые

из записей клиента будут удалены из таблицы `sales`, скидка для этого клиента должна уменьшиться.

Еще один вариант использования триггера `AFTER DELETE` — удаление связанной информации из других таблиц после удаления записи из базовой таблицы. Например, вы можете установить триггер, который удаляет запись о клиенте, если записи о продажах с соответствующим `customer_id` будут удалены из таблицы `sales`. Запустите следующую команду для создания триггера:

```
1. DELIMITER //
2.
3. CREATE TRIGGER delete_related_info
4.
5. AFTER DELETE
6.
7. ON sales
8.
9. FOR EACH ROW
10.
11.   Delete from customers where customer_id=OLD.customer_id;//
12.
13.   DELIMITER ;
14.
```

Copy

Далее запустите следующую команду, чтобы удалить все записи о продажах, связанных с `customer_id`, равному `2`:

```
1. Delete from sales where customer_id='2';
2.
```

Copy

```
Output

Query OK, 1 row affected (0.00 sec)
```

Теперь проверьте, удалились ли записи для этого клиента из таблицы `sales`:

```
1. Select * from customers where customer_id='2';
2.
```

Вы получите вывод `Empty Set`, поскольку запись клиента, связанная с `customer_id` `2`, была удалена триггером:

```
Output

Empty set (0.00 sec)
```

Вы научились использовать все виды триггеров для выполнения разных функций. Далее вы узнаете, как удалить триггер из базы данных, если он вам больше не нужен.

Шаг 8 — Удаление триггеров

Как и любой другой объект базы данных, вы можете удалить триггеры с помощью команды `DROP`. Синтакс удаления триггера следующий:

```
Drop trigger [TRIGGER NAME];
```

Например, чтобы удалить последний созданный триггер `AFTER DELETE`, выполните следующую команду:

```
1. Drop trigger delete_related_info;
2.
```

```
Output

Query OK, 0 rows affected (0.00 sec)
```

Необходимость удаления триггеров возникает, когда вы хотите воссоздать его структуру. В таком случае вы можете сбросить триггер и создать новый с помощью разных команд для триггеров.

Порядок выполнения работы

Примеры создания триггеров:

1.Для примера возьмём таблицу **city** из БД **world** следующего вида:

Field	Type	Null	Key	Default	Extra
ID	int	NO	PRI	NULL	auto_increment
Name	char(35)	NO			
CountryCode	char(3)	NO	MUL		
District	char(20)	NO			
Population	int	NO		0	

Создадим таблицу в которой будут регистрироваться некоторые действия над таблицей **city**:

```
create table CityLog (
logDate date,
action varchar(255));
```

2. Создадим триггер, который после завершения обновления численности населения (**Population)** в таблице city, будет записывать в журнал соответствующую строку:

```
DELIMITER $$
create trigger logCityPopulationUpd after update on city for each row
begin
```

```
insert into CityLog (logDate, action) values (now(), 'Количество населения обновлено!');
end $$
```

Теперь, если мы выполним обновление численности населения в городе:

```
UPDATE `world`.`city`
SET `Population` = 5555 WHERE `ID` = 5;
```

То в таблице появиться соответствующая строка:

```
select * from CityLog;
```

	logDate	action
▶	2022-06-25	Количество населения обновлено!

3. Отредактируем структуру таблицы **CityLog**, чтобы фиксировать не только дату, но и время.

И создадим триггер, который фиксирует удаление данных из таблицы **city**:

```
DELIMITER $$
create trigger logCityPopulationDel after delete on city for each row
begin
insert into CityLog (action, logTime) values ('Произведено удаление', now());
end $$
```

```
select * from CityLog;
```

	Количество населения обновлено!	2022-06-25 20:43:02
	Произведено удаление	2022-06-25 20:43:21

4. Следующий триггер будет связан с таблицей city и будет вызываться перед вставкой записи для проверки поля Population. Функция триггера состоит в том, чтобы проверить, превышает ли значение Population, вставляемое в таблицу city, величину 10000, и выдать ошибку, если это так.

```
DELIMITER //
CREATE TRIGGER validate_population_amount
```

```
BEFORE INSERT
ON city
FOR EACH ROW
IF NEW.Population<10000 THEN
SIGNAL SQLSTATE '45000'
SET MESSAGE_TEXT = 'Количество меньше 10000 ';
END IF//
DELIMITER ;
```

Строки

```
SIGNAL SQLSTATE '45000'
SET MESSAGE_TEXT = 'Количество меньше 10000 ';
```

Будут сигнализировать об ошибке и информировать пользователя.

```
insert into city values (4080, 'Gomel', 'BY', 'Gomelskay', 900);
```

```
Error Code: 1644. Количество меньше 10000
```

5. Создадим таблицу, в которой будем фиксировать изменения при обновлении над данными в таблице city. Она имеет следующую структуру:

```
create table populationLog (
cityId int,
logdate datetime,
oldPopulation int,
newPopulation int);
```

В эту таблицу будем записывать ID изменяемой записи, дату и время изменения и старое значение поля и новое значение поля численности населения. Создадим соответствующий триггер:

```
DELIMITER //
create trigger LogPopulation after update
on city
```

```
for each row

begin

insert into populationLog values(NEW.ID, now(), OLD.Population,
NEW.Population);

END//

DELIMITER ;
```

Выполним команду обновления:

```
UPDATE `world`.`city`

SET `Population` = 55566666 WHERE `ID` = 5;
```

Просмотрим содержимое таблицы populationLog:

```
1 •   SELECT * FROM world.populationlog;
```

Result Grid | Filter Rows: | Export:

cityId	logdate	oldPopulation	newPopulation
5	2022-06-25 21:14:05	5556	55566666

Изменение зафиксировано в журнале.