# План занятия

1. Транзакции
2. Пользовательские функции
3. Хранимые процедуры
4. Представления
5. Триггеры

# Транзакции

В РСУБД транзакции – это набор операций, который должен выполниться полностью или невыполнить ни одной своей операции.

## Добавление нового пользователя

Когда нам нужно добавить полную инфу о пользователе, то проблема/особенность данной операции в необходимости внесения записей сразу в 2 таблицы (**users, profiles**). Соответственно, надо либо сделать все, либо – ничего (отменить начатое в случае невозможности завершить действие).

* Добавим нового пользователя (сразу в 2 таблицы)
* Заодно потренируемся в работе с переменными

|  |
| --- |
| -- Транзакция по добавлению нового пользователя  **START** **TRANSACTION**;  **INSERT** **INTO** users (firstname, lastname, email, phone)  **VALUES** ('New', 'User', 'new@mail.com', 454545456);  **SELECT** @last\_user\_id := (**SELECT** **MAX**(id) **FROM** users); -- опасный способ  **INSERT** **INTO** profiles (user\_id, gender, birthday, hometown)  **VALUES** (@last\_user\_id, 'M', '1999-10-10', 'Moscow');    **COMMIT**; |
| * Результат пойдем искать в 2 таблицах: users, profiles * Должна была добавиться новая запись в кждоой из них |

## Откат транзакции

* Поменяем последнюю строку скрипта с **COMMIT** на **ROLLBACK**

|  |
| --- |
| -- Транзакция по добавлению нового пользователя  **START** **TRANSACTION**;  **INSERT** **INTO** users (firstname, lastname, email, phone)  **VALUES** ('New', 'User', 'new3@mail.com', 454545456);  **SELECT** @last\_user\_id := (**SELECT** id **FROM** users **WHERE** email = 'new@mail.com');  **INSERT** **INTO** profiles (user\_id, gender, birthday, hometown)  **VALUES** (@last\_user\_id, 'M', '1999-10-10', 'Moscow');    **ROLLBACK**; |
| * Скрипт исполнится без ошибок * Новых строк в таблицы добавлено не будет |

## Удаление пользователя

* Напишем транзакцию, удаляющую запись о пользователе из обеих таблиц (**users** и **profiles**)

|  |
| --- |
| -- Транзакция по удалению пользователя  **START** **TRANSACTION**;  **SELECT** @last\_user\_id := (**SELECT** **MAX**(id) **FROM** users); -- опасный способ  **delete** **from** profiles  **where** user\_id = @last\_user\_id;  **delete** **from** users  **where** id = @last\_user\_id;  **COMMIT**; |
| * Пользователя с почтой 'new@mail.com' больше нет в БД |

### Временно выключить проверку внешних ключей

Но если бы у удаляемого пользователя были посты, фотки, лайки, сообщения и т.д., то мы не смогли бы его удалить. В этом случае можно отключить проверку внешних ключей.

Это фича именно MySQL. В других СУБД возможность отключения проверки внешних ключей реализована по-другому (синтаксис отличается).

* Отключим проверку внешних ключей

SET FOREIGN\_KEY\_CHECKS = 0;

* Выполним удаление

|  |
| --- |
| **delete** **from** users  **where** id = ... |

* Включим обратно проверку ключей

SET FOREIGN\_KEY\_CHECKS = 1;

* Посмотреть текущее значение этой переменной можно так:

|  |
| --- |
| -- посмотреть значение переменной foreign\_key\_checks  **SELECT** @@**GLOBAL**.foreign\_key\_checks, @@**SESSION**.foreign\_key\_checks; |
|  |

### Изменить тип внешнего ключа

Другим путем решения проблемы с внешними ключами является изменение его типа на менее строгий.

Внешний ключ может иметь разную логику поведения при удалении основной записи (на которую он ссылается):

|  |
| --- |
| *reference\_option*:  RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT |

* Попробуем удалить сообщество, в котором есть члены

|  |
| --- |
| -- НЕ сможем удалить  **delete** **from** communities  **where** id = 10; |
| * Не можем этого сделать из-за ограничений внешнего ключа |

Поменяем логику поведения внешнего ключа при удалении записей с запрета (**RESTRICT**) на каскадное удаление (**CASCADE**). В этом случае нам необходимо:

* Удалить существущий внешний ключ

|  |
| --- |
| -- удалим внешний ключ  **ALTER** **TABLE** vk.users\_communities  **DROP** **FOREIGN** **KEY** <fk\_name>; |
|  |

* Заново его добавить с требуемой новой логикой работы

|  |
| --- |
| -- создадим заново внешний ключ с правилом каскадного удаления зависимых записей  **ALTER** **TABLE** vk.users\_communities  **ADD** **CONSTRAINT** users\_communities\_fk  **FOREIGN** **KEY** (community\_id)  **REFERENCES** vk.communities(id) **ON** **DELETE** **cascade**; |
|  |

* Теперь сможем удалить любое сообщество (но вместе с его членами)

|  |
| --- |
| -- сможем удалить и сразу удалим зависимые записи  **delete** **from** communities  **where** id = 10; |
| * Сообщество удалено успешно * Инфа о членстве пользователей в этом сообществе тоже удалена (табл. **users\_communities**) |

# Функции

## Описание задачи

Напишем функцию, которая будет считать коэффициент направленности дружбы.

Формула: Кол-во приглашений в друзья к пользователю / (разделим на) Кол-во заявок в друзья от пользователя.

Из результата (которым будет некоторое вещественное число) можно будет делать некоторые выводы:

1. Чем больше значение, тем популярность пользователя выше.
2. Если значение меньше единицы - пользователь инициатор связей (и наоборот).

## Код фукнции

* Наша функция будет только читать данные => **READS** **SQL** **DATA**

|  |
| --- |
| /\*  Направленность дружбы  Кол-во приглашений в друзья к пользователю  /  Кол-во приглашений в друзья от пользователя  Чем больше - популярность выше  Если значение меньше единицы - пользователь инициатор связей.  \*/  USE vk;  **DROP** **FUNCTION** **IF** **EXISTS** friendship\_direction;  DELIMITER // -- выставим разделитель  **CREATE** **FUNCTION** friendship\_direction(check\_user\_id **INT**)  **RETURNS** **FLOAT** **READS** **SQL** **DATA**  **BEGIN**  -- объявим переменные  **DECLARE** requests\_to\_user **INT**;  **DECLARE** requests\_from\_user **INT**;    -- получим запросы к пользователю  **SET** requests\_to\_user =  (**SELECT** **COUNT**(\*)  **FROM** friend\_requests  **WHERE** target\_user\_id = check\_user\_id);    -- получим запросы от пользователя  **SET** requests\_from\_user =  (**SELECT** **COUNT**(\*)  **FROM** friend\_requests  **WHERE** initiator\_user\_id = check\_user\_id);    -- разделим первое на второе и вернем результат  **RETURN** requests\_to\_user / requests\_from\_user;  **END**// -- не забываем наш новый разделитель  DELIMITER ; -- вернем прежний разделитель |

### Опции при создании функции

Офф. дока: <https://dev.mysql.com/doc/refman/8.0/en/stored-programs-logging.html>

## Вызов функции / результаты

* Из результатов видно, что пользователь чаще сам инициатор дружбы

|  |
| --- |
| **SELECT** friendship\_direction(1); |
|  |

* Округлим результат с помощью функции **TRUNCATE**

|  |
| --- |
| **SELECT** TRUNCATE(friendship\_direction(1), 2); |
|  |

* Посчитаем результат для другого пользоваетля (**id** = 11)
* Из результатов видно, что пользователь достаточно популярен у других участников сети.

|  |
| --- |
| **SELECT** TRUNCATE(friendship\_direction(11), 2); |
|  |

# Процедуры

## Описание задачи

Напишем процедуру, которая будет предлагать пользователям новых друзей.

Критерии выбора пользователей:

* из одного города
* состоят в одной группе
* друзья друзей

Из выборки будем показывать 5 человек в случайной комбинации.

## Код процедуры

|  |
| --- |
| **drop** **procedure** **if** **exists** friendship\_offers;  delimiter //  **create** **procedure** friendship\_offers(**in** for\_user\_id **INT**)  **begin**  -- общий город  **select** p2.user\_id  **from** profiles p1  **join** profiles p2  **on** p1.hometown = p2.hometown  **where** p1.user\_id = for\_user\_id  **and** p2.user\_id <> for\_user\_id -- исключим себя    **union**    -- общие группы  **select** uc2.user\_id  **from** users\_communities uc1  **join** users\_communities uc2  **on** uc1.community\_id = uc2.community\_id  **where** uc1.user\_id = for\_user\_id  **and** uc2.user\_id <> for\_user\_id -- исключим себя  **union**    -- друзья друзей  -- получим друзей друзей  -- объединяем таблицу саму с собой 3 раза  -- фильтруем «первую» таблицу по for\_user\_id  **select** fr3.target\_user\_id  **from** friend\_requests fr1  **join** friend\_requests fr2  **on** (fr1.target\_user\_id = fr2.initiator\_user\_id  **or** fr1.initiator\_user\_id = fr2.target\_user\_id)  **join** friend\_requests fr3  **on** (fr3.target\_user\_id = fr2.initiator\_user\_id  **or** fr3.initiator\_user\_id = fr2.target\_user\_id)  **where** (fr1.initiator\_user\_id = for\_user\_id **or** fr1.target\_user\_id = for\_user\_id)  **and** fr2.status = 'approved' -- оставляем только подтвержденную дружбу  **and** fr3.status = 'approved'  **and** fr3.target\_user\_id <> for\_user\_id -- исключим себя    **order** **by** rand() -- будем брать всегда случайные записи  **limit** 5; -- ограничим всю выборку до 5 строк  **END**// -- не забываем наш новый разделитель  DELIMITER ; -- вернем прежний разделитель |

## Вызов продедуры / результаты

* Каждый раз при вызове процедуры с одним и тем же параметром мы видим разный результат.

|  |
| --- |
| **CALL** friendship\_offers(1); |
|  |

# Представления

## Синтаксис

Офф. Дока:

<https://dev.mysql.com/doc/refman/8.0/en/create-view.html>

|  |
| --- |
| CREATE  [OR REPLACE]  [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]  [DEFINER = *user*]  [SQL SECURITY { DEFINER | INVOKER }]  VIEW *view\_name* [(*column\_list*)]  AS *select\_statement*  [WITH [CASCADED | LOCAL] CHECK OPTION] |

Использование алгоритма **MERGE** требует соответствия 1 к 1 между строками таблицы и основанного на ней представления. Алгоритм **TEMPTABLE** заносит содержимое представления во временную таблицу

**WITH CHECK OPTION** — при использовании данной конструкции все добавляемые или изменяемые строки будут проверяться на соответствие определению представления. В случае несоответствия данное изменение не будет выполнено. Обратите внимание, что при указании данной конструкции для необновляемого представления возникнет ошибка и представление не будет создано. (подробнее речь об этом пойдет ниже).

## Пример представления с БД vk

* Напишем представление, выбирающее друзей пользователя

|  |
| --- |
| -- представление, выбирающее друзей пользователя  **CREATE** **or** **replace** **VIEW** view\_friends  **AS**  **select** \*  **FROM** users u  **JOIN** friend\_requests fr **ON** u.id = fr.target\_user\_id  **WHERE**  fr.status = 'approved'  **union**    **select** \*  **FROM** users u  **JOIN** friend\_requests fr **ON** u.id = fr.initiator\_user\_id  **WHERE**  fr.status = 'approved' |
|  |

* Теперь можно обращаться к представлению следующим образом

|  |
| --- |
| -- получим друзей пользователя из представления  **select** \*  **from** view\_friends  **where** id = 1 |
|  |

## Обновление данных через представление

Представление называется обновляемым, если к нему могут быть применимы операторы UPDATE и DELETE для изменения данных в таблицах, на которых основано представление. Для того, чтобы представление было обновляемым должно быть выполнено 2 условия:

1. Соответствие 1 к 1 между строками представления и таблиц, на которых основано представление, т.е. каждой строке представления должно соответствовать по одной строке в таблицах-источниках.
2. Поля представления должны быть простым перечислением полей таблиц, а не выражениеями col1/col2 или col1+2.

## Удаление представления

* Для удаления представления используется команда **DROP VIEW**

|  |
| --- |
| -- удаление представления  **drop** **view** view\_friends; |
|  |

# Триггеры

Чтобы в СУБД запускались действия по событиям (например, после удаления данных в определенной таблице), используются триггеры.

ENG docs:

<https://dev.mysql.com/doc/refman/8.0/en/create-trigger.html>

RUS docs:

Отсутствует офф. Дока

## Синтаксис

|  |
| --- |
| CREATE  [DEFINER = *user*]  TRIGGER *trigger\_name*  *trigger\_time* *trigger\_event*  ON *tbl\_name* FOR EACH ROW  [*trigger\_order*]  *trigger\_body*  *trigger\_time*: { BEFORE | AFTER }  *trigger\_event*: { INSERT | UPDATE | DELETE }  *trigger\_order*: { FOLLOWS | PRECEDES } *other\_trigger\_name* |

Плюсы триггеров:

* Они очень удобны для обработки каких-либо событий в СУБД: будучи установленным один раз, обработчик будет работать и выполнять свою функцию.
* Главный плюс триггеров: они минуют долгую фазу анализа и компиляции запроса.

Недостатки триггеров:

* при отладке иногда достаточно сложно понять, что какие-то действия обрабатывает триггер. Поэтому многие считают, что это плохой паттерн, при его использовании отладка очень сложна, особенно если триггеров очень много.

Вывод: триггеры нельзя использовать бездумно и вешать без предварительного описания, а также не взвесив все за и против данного подхода.

## Для чего применяются

* Логгирование
* Вызов внешних сервисов
* Проверки данных
* Дополнительные вычисления
* Другая бизнес-логика

## Примеры с БД vk

* Напишем триггер для операции обновления профиля (дата рождения должна быть в прошлом)

|  |
| --- |
| -- триггер для проверки возраста пользователя перед обновлением  **DELIMITER //**  **CREATE** **TRIGGER** check\_user\_age\_before\_update **BEFORE** **UPDATE** **ON** profiles  **FOR** **EACH** **ROW**  **begin**  **IF** **NEW**.birthday >= **CURRENT\_DATE**() **THEN**  **SIGNAL** **SQLSTATE** '45000' **SET** MESSAGE\_TEXT = 'Update Canceled. Birthday must be in the past!';  **END** **IF**;  **END**//  **DELIMITER ;** |
|  |

* Напишем триггер для корректировки возраста пользователя

|  |
| --- |
| -- триггер для корректировки возраста пользователя при вставке новых строк  **drop** **TRIGGER** **if** **exists** check\_user\_age\_before\_insert;  **DELIMITER //**  **CREATE** **TRIGGER** check\_user\_age\_before\_insert **BEFORE** **INSERT** **ON** profiles  **FOR** **EACH** **ROW**  **begin**  **IF** **NEW**.birthday > **CURRENT\_DATE**() **THEN**  **SET** **NEW**.birthday = **CURRENT\_DATE**();  **END** **IF**;  **END**//  **DELIMITER ;** |
|  |

* Проверим работу триггера обновления

|  |
| --- |
| -- обновим ДР пользователя с некорректной датой  **UPDATE** profiles  **SET** birthday='2020.10.10'  **WHERE** user\_id = 1; |
| * Видим наше сообщение об ошибке |

* Проверим работу триггера на вставку
* Удалим профиль с user\_id = 3
* Добавим новый профиль с user\_id = 3 и ДР в будущем и сохраним

|  |
| --- |
|  |
| * Вставилась запись с сегодняшней датой |